Understanding ECMAScript 6

```
class Square extends Rectangle {

    constructor(size) {
        this.length = size;
        this.width = size;
    }

    toString() {
        return "[Square" + this.length + "x" + this.wi
    }

}
```

Nicholas C. Zakas

目錄

关于

原书《Understanding ECMAScript 6》，作者 Nicholas C. Zakas ，在线阅读地址。此书为开源书籍，可访问它的github 仓库，若发现问题可以去提交 issue 。

此书的官方中文版已经在 2017 年 7 月由电子工业出版社正式出版，可查看亚马逊上的购买链接。而此处的翻译只是我本人的义务翻译。

此书中文版出版之前， **oshotokill** 对本书进行了义务翻译（阅读地址），但整体工作未完成，欠缺三章：

- 第六章 符号与符号属性
- 第十章 增强的数组功能
- 第十二章 代理与反射接口

原先我只是对 **oshotokill** 的翻译提出了修正建议，此后才对这三章进行了翻译。最后我又决定将此书完整重译一遍，只遗漏了引言部分。其中有少数地方借鉴了 **oshotokill** 的翻译，特此表示感谢。

在 **GitBook** 页面上阅读本书，请点击首页右上角的**“Read”**按钮；也可以点击**“Download”**按钮下载 **PDF** 、 **Mobi** 或 **ePub** 格式的电子书。翻译文字的修正过程可查阅 修改记录。

翻译过程确实比较累，毕竟是第一次做全书翻译。这次是出于本人兴趣的义务翻译，可能存在错误，肯定也会有语言表述方面的一些问题，欢迎指正。无论是对原书内容的指正还是对译文的指正，都可以。

本书原作者 Zakas 长期供职于雅虎，是著名的 JS 库 YUI 的主要作者，有着非常丰富的一线工作经验。他同时也是一个成功的作者，其最重要的著作《 JavaScript 高级编程》基本上是 JS 领域的必读之作，而他还出版了另一些质量很高的著作。《高级编程》一书实际上并不是完全高深的内容，而是从基本的层次开始讲述，逐步提高，全书结构比较良好，对初学者或有一定经验的开发者来说都是很有用的。

ECMAScript 6 作为 JS 的新标准 2015 年便已推出，已经不是最新版了，但到目前为止市面上完整介绍其特性的书却非常少。其中原因也许是语法变动太大，而有些特性浏览器直到现在都没有完全支持。不过通过 Babel 之类的工具，早就可以开始使用 ES6 了，所以这方面完整著作的缺失不能不说是一个遗憾。

本书的英文版在 2016 年 8 月 30 日正式出版，一如既往保持了 Zakas 的一贯水准。组织结构比较合理，并不完全是罗列新特性，而是有侧重点地进行介绍。同时在介绍某些特性时，还会特别提醒读者其中的范例运行效率不高，可考虑用其他方式实现，体现出作者负责任的态度。此外，在不少地方还会讲述新标准的制定背景，有的是因为原有功能缺乏，有的是旧版

ES 有坑容易误踩，因此有些新标准才顺应形势得以出台。如果认真学习此书，不但对了解并使用 ES6 有帮助，也会有助于规避 JS 的一些旧坑。最后，此书在附录部分对 ES2016 也作了介绍。

即使像《JS 高级编程》这样的经典著作，也会存在一些问题。例如原书对于闭包的定义我个人就觉得很不满意。其定义不能说是错的，但有两个问题：1、用词有二义性；2、太过简略，没有在定义中体现出 JS 闭包的真正特征。

相应的，本书也存在一些问题。

1. 代码或引用内容中存在一些笔误。
2. 有些描述不符合浏览器的实际情况，这也许是浏览器对规范标准的支持有偏差。
3. 有少数错误，这在译文中都有标注，但可能还有译者所未发现的。同时因为翻译全书确实比较累，所以有些范例代码我比较快地跳过去了，没有特别仔细看。如果读者发现有错，可以在此处提出，也可以到原作者的 github 上去提交 issue （但是原作者回应可能不会太及时）。
4. 有些内容讲述得不够完整，例如 Promise 链的问题。在比较长的链中，如果中间抛出了一个错误，而这个错误没有被 `catch()` 及时捕获，就会沿着整个链继续向下传递，跳过链中的所有 `then()` 处理，直到遇到 `catch()` 为止，或是静默失败（链条下方没有任何拒绝处理的情况下）。这一点在原书中并没有明确进行描述，这是让我觉得美中不足的地方。而且关于 Promise 链，原书的范例都太简单了，除上述所提的内容外，其实还有更多可讲的。

但瑕不掩瑜，这本书还是值得一看的好书。

引言

The JavaScript core language features are defined in a standard called ECMA-262. The language defined in this standard is called ECMAScript. What you know as JavaScript in browsers and Node.js is actually a superset of ECMAScript. Browsers and Node.js add more functionality through additional objects and methods, but the core of the language remains as defined in ECMAScript. The ongoing development of ECMA-262 is vital to the success of JavaScript as a whole, and this book covers the changes brought about by the most recent major update to the language: ECMAScript 6.

通往 ES6 之路

In 2007, JavaScript was at a crossroads. The popularity of Ajax was ushering in a new age of dynamic web applications, while JavaScript hadn't changed since the third edition of ECMA-262 was published in 1999. TC-39, the committee responsible for driving the ECMAScript development process, put together a large draft specification for ECMAScript 4. ECMAScript 4 was massive in scope, introducing changes both small and large to the language. Updated features included new syntax, modules, classes, classical inheritance, private object members, optional type annotations, and more.

The scope of the ECMAScript 4 changes caused a rift to form in TC-39, with some members feeling that the fourth edition was trying to accomplish too much. A group of leaders from Yahoo, Google, and Microsoft created an alternate proposal for the next version of ECMAScript that they initially called ECMAScript 3.1. The "3.1" was intended to show that this was an incremental change to the existing standard.

ECMAScript 3.1 introduced very few syntax changes, instead focusing on property attributes, native JSON support, and adding methods to already-existing objects. Although there was an early attempt to reconcile ECMAScript 3.1 and ECMAScript 4, this ultimately failed as the two camps had difficulty with the very different perspectives on how the language should grow.

In 2008, Brendan Eich, the creator of JavaScript, announced that TC-39 would focus its efforts on standardizing ECMAScript 3.1. They would table the major syntax and feature changes of ECMAScript 4 until after the next version of ECMAScript was standardized, and all members of the committee would work to bring the best pieces of ECMAScript 3.1 and 4 together after that point into an effort initially nicknamed ECMAScript Harmony.

ECMAScript 3.1 was eventually standardized as the fifth edition of ECMA-262, also described as ECMAScript 5. The committee never released an ECMAScript 4 standard to avoid confusion with the now-defunct effort of the same name. Work then began on ECMAScript Harmony, with ECMAScript 6 being the first standard released in this new “harmonious” spirit.

ECMAScript 6 reached feature complete status in 2015 and was formally dubbed “ECMAScript 2015.” (But this text still refers to it as ECMAScript 6, the name most familiar to developers.) The features vary widely from completely new objects and patterns to syntax changes to new methods on existing objects. The exciting thing about ECMAScript 6 is that all of its changes are geared toward solving problems that developers actually face.

关于本书

A good understanding of ECMAScript 6 features is key for all JavaScript developers going forward. The language features introduced in ECMAScript 6 represent the foundation upon which JavaScript applications will be built for the foreseeable future. That’s where this book comes in. My hope is that you’ll read this book to learn about ECMAScript 6 features so that you’ll be ready to start using them as soon as you need to.

浏览器与 **Node.js** 的兼容性

Many JavaScript environments, such as web browsers and Node.js, are actively working on implementing ECMAScript 6. This book doesn’t attempt to address the inconsistencies between implementations and instead focuses on what the specification defines as the correct behavior. As such, it’s possible that your JavaScript environment may not conform to the behavior described in this book.

本书读者对象

This book is intended as a guide for those who are already familiar with JavaScript and ECMAScript 5. While a deep understanding of the language isn’t necessary to use this book, it will help you understand the differences between ECMAScript 5 and 6. In particular, this

book is aimed at intermediate-to-advanced JavaScript developers programming for a browser or Node.js environment who want to learn about the latest developments in the language.

This book is not for beginners who have never written JavaScript. You will need to have a good basic understanding of the language to make use of this book.

概述

Each of this book's thirteen chapters covers a different aspect of ECMAScript 6. Many chapters start by discussing problems that ECMAScript 6 changes were made to solve, to give you a broader context for those changes, and all chapters include code examples to help you learn new syntax and concepts.

Chapter 1: How Block Bindings Work talks about `let` and `const`, the block-level replacement for `var`.

Chapter 2: Strings and Regular Expressions covers additional functionality for string manipulation and inspection as well as the introduction of template strings.

Chapter 3: Functions in ECMAScript 6 discusses the various changes to functions. This includes the arrow function form, default parameters, rest parameters, and more.

Chapter 4: Expanded Object Functionality explains the changes to how objects are created, modified, and used. Topics include changes to object literal syntax, and new reflection methods.

Chapter 5: Destructuring for Easier Data Access introduces object and array destructuring, which allow you to decompose objects and arrays using a concise syntax.

Chapter 6: Symbols and Symbol Properties introduces the concept of symbols, a new way to define properties. Symbols are a new primitive type that can be used to obscure (but not hide) object properties and methods.

Chapter 7: Sets and Maps details the new collection types of `Set`, `WeakSet`, `Map`, and `WeakMap`. These types expand on the usefulness of arrays by adding semantics, de-duping, and memory management designed specifically for JavaScript.

Chapter 8: Iterators and Generators discusses the addition of iterators and generators to the language. These features allow you to work with collections of data in powerful ways that were not possible in previous versions of JavaScript.

Chapter 9: Introducing JavaScript Classes introduces the first formal concept of classes in JavaScript. Often a point of confusion for those coming from other languages, the addition of class syntax in JavaScript makes the language more approachable to others and more

concise for enthusiasts.

Chapter 10: Improved Array Capabilities details the changes to native arrays and the interesting new ways they can be used in JavaScript.

Chapter 11: Promises and Asynchronous Programming introduces promises as a new part of the language. Promises were a grassroots effort that eventually took off and gained in popularity due to extensive library support. ECMAScript 6 formalizes promises and makes them available by default.

Chapter 12: Proxies and the Reflection API introduces the formalized reflection API for JavaScript and the new proxy object that allows you to intercept every operation performed on an object. Proxies give developers unprecedented control over objects and, as such, unlimited possibilities for defining new interaction patterns.

Chapter 13: Encapsulating Code with Modules details the official module format for JavaScript. The intent is that these modules can replace the numerous ad-hoc module definition formats that have appeared over the years.

Appendix A: Smaller ECMAScript 6 Changes covers other changes implemented in ECMAScript 6 that you'll use less frequently or that didn't quite fit into the broader major topics covered in each chapter.

Appendix B: Understanding ECMAScript 7 (2016) describes the two additions to the standard that were implemented for ECMAScript 7, which didn't impact JavaScript nearly as much as ECMAScript 6.

排版约定

The following typographical conventions are used in this book:

- **Italics** introduces new terms
- `Constant width` indicates a piece of code or filename

Additionally, longer code examples are contained in constant width code blocks such as:

```
function doSomething() {
    // empty
}
```

Within a code block, comments to the right of a `console.log()` statement indicate the output you'll see in the browser or Node.js console when the code is execute, for example:

```
console.log("Hi");      // "Hi"
```

If a line of code in a code block throws an error, this is also indicated to the right of the code:

```
doSomething();          // error!
```

帮助与支持

You can file issues, suggest changes, and open pull requests against this book by visiting: https://github.com/nzakas/understandinges6

If you have questions as you read this book, please send a message to my mailing list: http://groups.google.com/group/zakasbooks.

致谢

Thanks to Jennifer Griffith-Delgado, Alison Law, and everyone at No Starch Press for their support and help with this book. Their understanding and patience as my productivity slowed to a crawl during my extended illness is something I will never forget.

I'm grateful for the watchful eye of Juriy Zaytsev as tech editor and to Dr. Axel Rauschmayer for his feedback and several conversations that helped to clarify some of the concepts discussed in this book.

Thanks to everyone who submitted fixes to the version of this book that is hosted on GitHub: ShMcK, Ronen Elster, Rick Waldron, blacktail, Paul Salaets, Lonniebiz, Igor Skuhar, jakub-g, David Chang, Kevin Sweeney, Kyle Simpson, Peter Bakondy, Philip Borisov, Shaun Hickson, Steven Foote, kavun, Dan Kielp, Darren Huskie, Jakub Narębski, Jamund Ferguson, Josh Lubaway, Marián Rusnák, Nikolas Poniros, Robin Pokorný, Roman Lo, Yang Su, alexyans, robertd, 404, Aaron Dandy, AbdulFattah Popoola, Adam Richeimer, Ahmad Ali, Aleksandar Djindjic, Arjunkumar, Ben Regenspan, Carlo Costantini, Dmitri Suvorov, Kyle Pollock, Mallory, Erik Sundahl, Ethan Brown, Eugene Zubarev, Francesco Pongiluppi, Jake Champion, Jeremy Caney, Joe Eames, Juriy Zaytsev, Kale Worsley, Kevin Lozandier, Lewis Ellis, Mohsen Azimi, Navaneeth Kesavan, Nick Bottomley, Niels Dequeker, Pahlevi Fikri Auliya, Prayag Verma, Raj Anand, Ross Gerbasi, Roy Ling, Sarbbottam Bandyopadhyay, and Shidhin.

Also, thanks to everyone who supported this book on Patreon: Casey Visco.

第一章 块级绑定

变量声明的工作机制历来是 JS 编程中最微妙的部分之一。在大多数类 C 语言中，变量（或绑定）总是在它被声明的地方创建。然而 JS 例外，变量实际创建的位置取决于你如何声明它，为此 ES6 提供了新特性以便能更好地控制变量的作用域。本章会演示传统的 `var` 声明为何会令人困惑，并介绍 ES6 的块级绑定，然后再给出一些相关的最佳实践。

var 声明与变量提升

使用 `var` 关键字声明的变量，无论其实际声明位置在何处，都会被视为声明于所在函数的顶部（如果声明不在任意函数内，则视为在全局作用域的顶部）。这就是所谓的变量提升（ **hoisting** ）。请参考如下函数定义来理解变量提升：

```
function getValue(condition) {

    if (condition) {
        var value = "blue";

        // 其他代码

        return value;
    } else {

        // value 在此处可访问，值为 undefined

        return null;
    }

    // value 在此处可访问，值为 undefined
}
```

如果你不太熟悉 JS ，或许会认为仅当 `condition` 的值为 `true` 时，变量 `value` 才会被创建。但实际上， `value` 无论如何都会被创建。 JS 引擎会默默地将 `getValue` 函数调整为如下形式：

```
function getValue(condition) {

    var value;

    if (condition) {
        value = "blue";

        // 其他代码

        return value;
    } else {

        return null;
    }
}
```

`value` 变量的声明被提升到了函数顶部，而初始化操作则保留在原处。这意味着在 `else` 分支内 `value` 变量也是可访问的，在此处它的值并未被初始化，因此是 `undefined` 。

JS 的初学者经常需要花点时间才能习惯变量提升，而若不理解这种特有行为，就可能导致程序 bug 。正因如此， ES6 引入了块级作用域，让变量的生命周期更加可控。

块级声明

块级声明也就是让所声明的变量在指定块的作用域外无法被访问。块级作用域（又被称为词法作用域）在如下情况被创建：

1. 在一个函数内部
2. 在一个由一对花括号包裹的代码块内部

块级作用域是很多类 C 语言的工作机制， ES6 引入块级声明，是为了给 JS 添加灵活性，并保证与其他语言的一致性。

let 声明

`let` 声明的语法与 `var` 的语法一致。大体上可以直接用 `let` 代替 `var` 进行变量声明，但会将变量的作用域限制在当前代码块中（其他细微差别会在稍后讨论）。由于 `let` 声明并不会被提升，因此若想让变量在整个代码块内部可用，需要手动将 `let` 声明放置到代码块顶部。此处有个范例：

```
function getValue(condition) {

    if (condition) {
        let value = "blue";

        // 其他代码

        return value;
    } else {

        // value 在此处不可用

        return null;
    }

    // value 在此处不可用
}
```

这种写法的 `getValue` 函数的行为更接近其他类 C 语言，合乎预期。由于变量 `value` 声明使用的是 `let` 而非 `var` ，该声明就没有被提升到函数定义的顶部，因此变量 `value` 在 `if` 代码块之外就无法访问；并且在 `condition` 的值为 false 时，该变量永远不会被声明并初始化。

禁止重复声明

如果一个标识符已经在代码块内部被定义，那么在此代码块内使用同一个标识符进行 `let` 声明就会导致错误。例如：

```
var count = 30;

// 语法错误
let count = 40;
```

在本例中， `count` 变量被声明了两次：一次使用 `var` ，另一次使用 `let` 。因为 `let` 不能在同一作用域内重复声明一个已有标识符，此处的 `let` 声明就会抛出错误。然而，在嵌套的作用域内使用 `let` 声明一个同名新变量，就不会有问题，以下代码对此进行了演示：

```
var count = 30;

// 不会抛出错误
if (condition) {

    let count = 40;

    // 其他代码
}
```

此处的 `let` 声明并没有抛出错误，这是由于它并不是在同一作用域级别再次创建 `count` 变量，而是在 `if` 语句内部创建新变量。在 `if` 代码块内部，这个新变量会屏蔽全局的 `count` 变量，从而在局部阻止对于后者的访问。

常量声明

在 ES6 中里也可以使用 `const` 语法进行声明。使用 `const` 声明的变量会被认为是常量（**constant**），意味着它们的值在被设置完成后就不允许再被更改。正因如此，所有的 `const` 变量都需要在声明时进行初始化，示例如下：

```
// 有效的常量
const maxItems = 30;

// 语法错误：未进行初始化
const name;
```

`maxItems` 变量在声明时被初始化，因此它的 `const` 声明是有效的。而 `name` 变量缺少了初始化操作，试图运行这段代码就会抛出错误。

对比常量声明与 **let** 声明

常量声明与 `let` 声明一样，都是块级声明。这意味着常量在声明它们的语句块之外是无法被访问的，并且声明也不会被提升，示例如下：

```
if (condition) {
    const maxItems = 5;

    // 其他代码
}

// maxItems 在此处无法访问
```

在此代码中，常量 `maxItems` 在 `if` 语句内被声明，随着语句执行完毕， `maxItems` 在代码块外部就不再可用。

`const` 声明与 `let` 的另一个相似点是：在同一作用域内定义一个已有变量时会抛出错误，无论是在全局还是函数作用域，无论该变量此前是用 `var` 声明的，还是用 `let` 声明的。例如以下代码：

```
var message = "Hello!";
let age = 25;

// 二者均会抛出错误
const message = "Goodbye!";
const age = 30;
```

两个 `const` 声明独立来看都是有效的，但在前面已存在 `var` 与 `let` 声明的情况下，它们都会出问题。

尽管有上述相似之处，但 `let` 与 `const` 之间仍有一个本质区别，试图对 `const` 声明的常量再次赋值会抛出错误，无论是否运行在严格模式下：

```
const maxItems = 5;

maxItems = 6;      // 抛出错误
```

与其他语言的常量类似， `maxItems` 变量不能被再次赋值。然而不同之处在于， JS 的常量若是一个对象，它所包含的值是可以被修改的。

使用 const 声明对象

`const` 声明会阻止对于变量绑定与变量自身值的修改，这意味着它并不会阻止对变量成员的修改。例如：

```
const person = {
    name: "Nicholas"
};

// 工作正常
person.name = "Greg";

// 抛出错误
person = {
    name: "Greg"
};
```

此处 `person` 在初始化时被绑定为带有单个属性的对象。修改 `person.name` 并不会抛出错误，因为该操作只修改了 `person` 对象的成员，而没有修改 `person` 的绑定值。当代码试图为 `person` 对象自身赋值，即试图更改变量绑定时，就会导致错误。 `const` 在变量方面的微妙工作机制容易被误解，但仅需牢记： `const` 阻止的是对于变量绑定的修改，而不阻止对成员值的修改。

暂时性死区

使用 `let` 或 `const` 声明的变量，在达到声明位置之前都是无法访问的，试图访问会导致一个引用错误，即便所用的操作通常是安全的，例如使用 `typeof` 运算符。示例如下：

```
if (condition) {
    console.log(typeof value);  // 引用错误
    let value = "blue";
}
```

此处的 `value` 变量使用了 `let` 进行定义与初始化，但该语句永远不会被执行，因为声明之前的那行代码抛出了一个错误。出现该问题是因为 `value` 位于被 JS 社区称为暂时性死区（ **temporal dead zone** ， TDZ ）的区域内。该名称并未在 ECMAScript 规范中被明确命名，但经常被用于描述 `let` 或 `const` 声明的变量为何在声明位置之前无法被访问。本小节描述的是暂时性死区导致的声明位置的微妙之处，尽管此处使用的都是 `let` ，但替换为 `const` 也不例外。

当 JS 引擎扫描接下来的代码块并发现变量声明时，它会在处理 `var` 时将声明提升到函数或全局作用域的顶部，而处理 `let` 或 `const` 时，则会将声明放入暂时性死区。任何在暂时性死区内访问变量的企图都会导致“运行时”错误（runtime error）。只有执行到变量的声明语句时，该变量才会从暂时性死区内被移除，才可以安全使用。

使用 `let` 或 `const` 声明的变量都无法规避暂时性死区。而且正如上例所演示的，这甚至影响了通常安全的 `typeof` 运算符。不过可以在变量被定义的代码块之外对该变量使用 `typeof` ，尽管其结果可能并非预期。参考如下代码：

```
console.log(typeof value);     // "undefined"

if (condition) {
    let value = "blue";
}
```

`typeof` 运算符被用于 `value` 变量被定义的代码块外部，此时 `value` 并未在暂时性死区内。这意味着 `value` 变量绑定尚不存在，而 `typeof` 仅会单纯返回 `"undefined"` 。

暂时性死区只是块级绑定的一个独特表现，而另一个独特表现则是在循环内使用它。

循环中的块级绑定

开发者或许最期望在 `for` 循环内使用块级作用域，也就是想让一次性的循环计数器仅能在循环内部使用。如下代码在 JS 中并不罕见：

```
for (var i = 0; i < 10; i++) {
    process(items[i]);
}

// i 在此处仍然可被访问
console.log(i);                     // 10
```

在其他默认使用块级作用域的语言中，这个例子能够照预期工作，也就是只有 `for` 语句才能访问变量 `i` 。然而在 JS 中，由于 `var` 声明导致了变量提升，循环结束后 `i` 仍然可被访问。若像如下代码那样改用 `let` ，就会看到预期行为：

```
for (let i = 0; i < 10; i++) {
    process(items[i]);
}

// i 在此处不可访问，抛出错误
console.log(i);
```

本例中的变量 `i` 仅在 `for` 循环内部可用，一旦循环结束，该变量在任意位置都不可访问。

循环内的函数

长期以来， `var` 的特点使得循环变量在循环作用域之外仍然可被访问，于是在循环内创建函数就会产生问题。考虑如下代码：

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push(function() { console.log(i); });
}

funcs.forEach(function(func) {
    func();     // 输出数值 "10" 十次
});
```

你原本可能预期这段代码会输出 0 到 9 的数值，但它却在同一行将数值 10 输出了十次。这是因为变量 `i` 在循环的每次迭代中都被共享了，意味着循环内创建的那些函数都拥有对于同一变量的引用。在循环结束后，变量 `i` 的值会是 `10` ，因此当 `console.log(i)` 被调用时，每次都打印出 `10` 。

为了修正这个问题，开发者在循环内使用立即调用函数表达式（IIFEs），以便在每次迭代中强制创建变量的一个新副本，示例如下：

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
        }
    }(i)));
}

funcs.forEach(function(func) {
    func();     // 从 0 到 9 依次输出
});
```

这种写法在循环内使用了 IIFE 。变量 `i` 被传递给 IIFE ，从而创建了作为副本的 `value` 变量，每个 `value` 都存储着每次迭代时变量 `i` 的值。迭代中的函数使用这些值副本，在循环从 0 到 9 的过程中每次都能输出预期结果。幸运的是，使用 `let` 与 `const` 的块级绑定可以在 ES6 中简化这种写法。

循环内的 let 声明

`let` 声明通过有效模仿上例中 IIFE 的作用而简化了循环。在每次迭代中，都会创建一个新的同名变量并对其进行初始化。这意味着你可以完全省略 IIFE 而获取预期的结果，就像这样：

```
var funcs = [];

for (let i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}

funcs.forEach(function(func) {
    func();     // 从 0 到 9 依次输出
})
```

与使用 `var` 声明配合 IIFE 相比，这里代码能达到相同效果，但无疑更加简洁。在循环中 `let` 声明每次都创建了一个新的 `i` 变量，并赋予了正确的值，于是在循环内部创建的函数都获得了各自的 `i` 副本。这种方式在 `for-in` 和 `for-of` 循环中同样适用，如下所示：

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();     // 依次输出 "a"、 "b"、 "c"
});
```

本例中的 `for-in` 循环的行为与 `for` 循环一致。每次循环都创建一个新的 `key` 变量绑定，因此每个函数都能够拥有它自身的 `key` 变量副本，结果每个函数都输出了各自不同的值。而若使用 `var` 来声明 `key` ，则所有函数都只会输出 `"c"` 。

> 需要着重了解的是： `let` 声明在循环内部的行为是在规范中特别定义的，与不提升变量声明的特征没有必然联系。事实上，在早期 `let` 的实现中并没有这种行为，后来才增加。

循环内的常量声明

ES6 规范没有明确禁止在循环中使用 `const` 声明，然而它会根据不同循环方式而有不同行为。在常规的 `for` 循环中，你可以在初始化时使用 `const` ，但循环会在试图改变该变量的值时抛出错误。例如：

```
var funcs = [];

// 在一次迭代后抛出错误
for (const i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
```

在此代码中， `i` 被声明为一个常量。循环的第一次迭代成功执行，此时 `i` 的值为 0 。在 `i++` 执行时，一个错误会被抛出，因为该语句试图更改常量的值。因此，在循环中你只能使用 `const` 来声明一个确定不会被更改的变量。

而另一方面， `const` 变量在 `for-in` 或 `for-of` 循环中使用时，与 `let` 变量效果相同。因此下面代码不会导致出错：

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

// 不会导致错误
for (const key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();     // 依次输出 "a"、 "b"、 "c"
});
```

这段代码与“循环内的 let 声明”小节的第二个例子几乎完全一样，唯一的区别是 `key` 的值在循环内不能被更改。 `const` 能够在 `for-in` 与 `for-of` 循环内工作，是因为循环为每次迭代创建了一个新的变量绑定，而不是像 `for` 循环那样试图去修改已绑定的变量的值。

全局块级绑定

`let` 与 `const` 不同于 `var` 的另一个方面是在全局作用域上的表现。当在全局作用域上使用 `var` 时，它会创建一个新的全局变量，并成为全局对象（在浏览器中是 `window` ）的一个属性。这意味着使用 `var` 可能会无意中覆盖一个已有的全局属性，就像这样：

```
// 在浏览器中
var RegExp = "Hello!";
console.log(window.RegExp);     // "Hello!"

var ncz = "Hi!";
console.log(window.ncz);        // "Hi!"
```

尽管全局的 `RegExp` 是定义在 `window` 上的，它仍然不能防止被 `var` 重写。这个例子声明了一个新的全局变量 `RegExp` 而覆盖了原有对象。类似的， `ncz` 定义为全局变量后就立即成为了 `window` 的一个属性。这就是 JS 通常的工作方式。

然而若你在全局作用域上使用 `let` 或 `const` ，虽然在全局作用域上会创建新的绑定，但不会有任何属性被添加到全局对象上。这也就意味着你不能使用 `let` 或 `const` 来覆盖一个全局变量，你只能将其遮蔽。此处有个范例：

```
// 在浏览器中
let RegExp = "Hello!";
console.log(RegExp);                    // "Hello!"
console.log(window.RegExp === RegExp);  // false

const ncz = "Hi!";
console.log(ncz);                       // "Hi!"
console.log("ncz" in window);           // false
```

此代码的 `let` 声明创建了 `RegExp` 的一个绑定，并遮蔽了全局的 `RegExp` 。这表示 `window.RegExp` 与 `RegExp` 是不同的，因此全局作用域没有被污染。同样， `const` 声明创建了 `ncz` 的一个绑定，但并未在全局对象上创建属性。若你不想在全局对象上创建属性，使用 `let` 与 `const` 会更安全。

> 若想让代码能从全局对象中被访问，你仍然需要使用 `var` 。这种做法常见于在浏览器中跨越帧或窗口去访问代码的场合。

块级绑定新的最佳实践

在 ES6 的开发阶段，被广泛认可的变量声明方式是：默认情况下应当使用 `let` 来代替 `var` 。对于多数 JS 开发者来说， `let` 的行为方式正是 `var` 本应有的方式，因此直接用 `let` 替代 `var` 更符合逻辑。在这种情况下，需要受到保护的变量再使用 `const` 。

然而，随着更多的开发者迁移到 ES6 上，一种替代方案变得更为流行：在默认情况下使用 `const` ，仅当明确变量值需要被更改的情况下才使用 `let` 。其理论依据是大部分变量在初始化之后都应当保持不变，因为预期外的改动是 bug 的源头之一。这种理念有着足够强大的吸引力，值得在你使用 ES6 之后依此进行实践。

总结

`let` 与 `const` 块级绑定将词法作用域引入了 JS 。这两种声明方式都不会进行提升，并且仅在声明它们的代码块内部可用。这样变量就能够在必要位置被准确声明，其表现更加接近其他语言，并且能减少无心之失。作为一个副作用，你不能在变量声明位置之前访问它们，即便使用的是 `typeof` 这样的安全运算符。由于块级绑定存在暂时性死区（ TDZ ），试图在声明位置之前访问它就会导致错误。

`let` 与 `const` 的表现在很多情况下都相似于 `var`，然而在循环中就有差异。在 `for-in` 与 `for-of` 循环中， `let` 与 `const` 都能在每一次迭代时创建一个新的绑定，这意味着在循环体内创建的函数可以使用当前迭代所绑定的循环变量值，而不是像使用 `var` 那样，统一使用循环结束时的变量值。在 `for` 循环中使用 `let` 声明时也是如此，不过在 `for` 循环中使用 `const` 声明则会导致错误。

块级绑定当前的最佳实践就是：在默认情况下使用 `const` ，而仅在变量值确实需要被更改的情况下才使用 `let` 。这在代码中能确保基本的不可变性，有助于防止某些类型的错误。

第二章 字符串与正则表达式

字符串可以说是编程中最重要的数据类型之一。它们在几乎所有高级语言中存在，有效运用便能编写出有用的程序。由此扩展出的正则表达式也十分重要，因为它们给了开发者更多操纵字符串的能力。基于这些事实， ES6 的创造者加强了字符串与正则表达式，为它们添加了新的能力，并补充了一些长期缺失的功能。本章会介绍这些变化。

更好的 Unicode 支持

在 ES6 之前， JS 的字符串以 16 位字符编码（ UCS-2 ）为基础。每个 16 位序列都是一个码元（ **code unit** ），用于表示一个字符。字符串所有的属性与方法都基于 16 位的码元，例如 `length` 属性与 `charAt()` 方法。当然， 16 位曾经足以容纳任何字符，然而由于 Unicode

引入了扩展字符集，情况就发生了变化。

> 译注：此段第一句话的原文是：
>
> > Before ES6, JavaScript strings revolved around 16-bit character encoding (UTF-16).
>
> 然而 UTF-16 是变长的字符编码方式，有 16 位与 32 位两种情况。 JS 原先使用的则是固定 16 位（双字节）的字符编码方式，即 UCS-2 。

UTF-16 码点

Unicode 的明确目标是给世界上每一个字符提供全局唯一标识符，而 16 位的字符长度限制已不能满足这种需求。这些全球唯一标识符被称为码点（ **code points** ），就是从 0 开始的数字。码点类似于字符代码，用一个数字来代表一个字符。字符编码要求将码点转换为内部一致的码元，而对于 UTF-16 来说，码点可以由多个码元组成。

在 UTF-16 中的第一个 2^{16} 码点表示单个 16 位码元，这个范围被称为多语言基本平面（ **Basic Multilingual Plane** ， BMP ）。任何超出该范围的码点都不能用单个 16 位码元表示，而是会落在扩展平面（ **supplementary planes** ）内。 UTF-16 引入了代理对（ **surrogate pairs** ）来解决这个问题，允许使用两个 16 位码元来表示单个码点。这意味着字符串内的任意单个字符都可以用一个码元（共 16 位）或两个码元（共 32 位）来表示，前者对应基本平面字符，而后者对应扩展平面字符。

在 ES5 中，所有字符串操作都基于 16 位码元，这表示在处理包含代理对的 UTF-16 字符时会出现预期外的结果，就像这个例子：

```
var text = "" ;

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(text.charAt(0));        // ""
console.log(text.charAt(1));        // ""
console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
```

这个 Unicode 字符 `""` 使用了代理对，因此，上面的 JS 字符串操作会将该字符串当作两个 16 位字符来对待，这意味着：

- `text` 的长度属性值是 2 ，而不是应有的 1 。
- 意图匹配单个字符的正则表达式匹配失败了，因为它认为这里有两个字符。
- `charAt()` 方法无法返回一个有效的字符，因为这里每 16 位码点都不是一个可打印字符。

`charCodeAt()` 方法同样无法正确识别该字符，它只能返回每个码元的 16 位数字，但在 ES5 中，这已经是对 `text` 变量所能获取到的最精确的值了。

ES6 则是强制执行 UTF-16 字符编码以解决此类问题，从而让字符串操作标准化，也就意味着 JS 能够支持专门针对代理对的功能设计。本章接下来会讨论与此有关的几个典型案例。

codePointAt() 方法

ES6 为全面支持 UTF-16 而新增的方法之一是 `codePointAt()` ，它可以在给定字符串中按位置提取 Unicode 码点。该方法接受的是码元位置而非字符位置，并返回一个整数值，就像下面的 `console.log()` 范例所展示的：

```
var text = "a" ;

console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
console.log(text.charCodeAt(2));    // 97

console.log(text.codePointAt(0));   // 134071
console.log(text.codePointAt(1));   // 57271
console.log(text.codePointAt(2));   // 97
```

`codePointAt()` 方法的返回值一般与 `charCodeAt()` 相同，除非操作对象并不是 BMP 字符。 `text` 字符串的第一个字符不是 BMP 字符，因此它占用了两个码元，意味着该字符串的 `length` 属性是 3 而不是 2 。 `charCodeAt()` 方法只返回了位置 0 的第一个码元；而 `codePointAt()` 返回的是完整的码点，即使该码点占用了多个码元。对于位置 1 （第一个字符的第二个码元）和位置 2 （ `"a"` 字符）来说，两个方法返回的值则是相同的。

判断字符包含了一个还是两个码元，对该字符调用 `codePointAt()` 方法就是最简单的方法。可以照下面的函数这么写：

```
function is32Bit(c) {
    return c.codePointAt(0) > 0xFFFF;
}

console.log(is32Bit("" ));          // true
console.log(is32Bit("a"));           // false
```

16 位字符的上边界用十六进制表示就是 `FFFF` ，因此任何大于该数字的码点必定用两个码元（共 32 位）来表示。

String.fromCodePoint() 方法

当 ECMAScript 提供了某种方法时，它一般也会给出方法来处理相反的操作。你可以使用 `codePointAt()` 来提取字符串内某个字符的码点，也可以借助 `String.fromCodePoint()` 用给定的码点来产生包含单个字符的字符串。例如：

```
console.log(String.fromCodePoint(134071));  // ""
```

可以将 `String.fromCodePoint()` 视为 `String.fromCharCode()` 的完善版本。二者处理 BMP 字符时会返回相同结果，只有处理 BMP 范围之外的字符时才会有差异。

normalize() 方法

Unicode 另一个有趣之处是，在排序或其它一些比较操作中，不同的字符可能会被认为是等同的。有两种方式可以定义这种关联性：第一种是规范相等性（ **canonical equivalence** ），意味着两个码点序列在所有方面都被认为是可互换的。例如，两个字符的组合可以按规范等同于另一个字符。第二种关联性是兼容性（ **compatibility** ），两个兼容的码点序列看起来有差别，但在特定条件下可互换使用。

由于这些关联性，两个包含不同码点序列的字符串，可能代表着相同的文本内容。例如，字符 `"æ"` 与双字符的字符串 `"ae"` 有时能互换使用，但它们并不严格相等，除非使用某种手段来标准化。

ES6 给字符串提供了 `normalize()` 方法，以支持 Unicode 标准形式。该方法接受单个可选的字符串参数，用于指示需要使用下列哪种 Unicode 标准形式：

- Normalization Form Canonical Composition (`"NFC"`)，这是默认值；
- Normalization Form Canonical Decomposition (`"NFD"`)；
- Normalization Form Compatibility Composition (`"NFKC"`)；
- Normalization Form Compatibility Decomposition (`"NFKD"`)。

解释这四种形式的差异超出了本书的范围。仅需记住，当进行字符串比较时，必须将它们标准化为同一种形式。例如：

```
var normalized = values.map(function(text) {
    return text.normalize();
});

normalized.sort(function(first, second) {
    if (first < second) {
        return -1;
    } else if (first === second) {
        return 0;
    } else {
        return 1;
    }
});
```

此代码将 `values` 数组中的字符串转换为一种标准形式，以便让转换后的数组可以被正确排序。你也可以在比较过程中调用 `normalize()` ，以便直接对原始数组进行排序。如下所示：

```
values.sort(function(first, second) {
    var firstNormalized = first.normalize(),
        secondNormalized = second.normalize();

    if (firstNormalized < secondNormalized) {
        return -1;
    } else if (firstNormalized === secondNormalized) {
        return 0;
    } else {
        return 1;
    }
});
```

此代码的重点在于： `first` 与 `second` 再一次被使用同一方式标准化了。这两个例子利用的是默认值，即 NFC ，不过你也能随便指定其他任意一种，就像这样：

```
values.sort(function(first, second) {
    var firstNormalized = first.normalize("NFD"),
        secondNormalized = second.normalize("NFD");

    if (firstNormalized < secondNormalized) {
        return -1;
    } else if (firstNormalized === secondNormalized) {
        return 0;
    } else {
        return 1;
    }
});
```

如果你之前从未担心过 Unicode 标准化方面的问题，那么可能暂时还不太会用到这个方法。然而若你曾经开发过国际化的应用，就一定会发现 `normalize()` 方法的价值。

新方法并不是 ES6 为 Unicode 字符串提供的唯一改进，新增的还有两个有用的语法要素。

正则表达式 u 标志

你可以使用正则表达式来完成字符串的很多通用操作。但要记住，正则表达式假定单个字符使用一个 16 位的码元来表示。为了解决这个问题， ES6 为正则表达式定义了用于处理 Unicode 的 `u` 标志。

u 标志实例

当一个正则表达式设置了 `u` 标志时，它的工作模式将切换到针对字符而非码元。这意味着正则表达式将不会被字符串中的代理对所混淆，而是会按预期工作。例如，研究以下代码：

```
var text = "" ;

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(/^.$/u.test(text));     // true
```

正则表达式 `/^.$/` 会匹配只包含单个字符的任意字符串。当不使用 `u` 标志时，该正则表达式只匹配码元，所以不能匹配由两个码元表示的这个日文字符。启用 `u` 标志后，正则表达式就会比较字符而不是码元，因此能匹配这个日文字符。

计算码点数量

可惜的是， ES6 并没有添加用于判断字符串包含多少个码点的方法，但借助 `u` 标志，可以使用正则表达式来进行计算，如下所示：

```
function codePointLength(text) {
    var result = text.match(/[\s\S]/gu);
    return result ? result.length : 0;
}

console.log(codePointLength("abc"));    // 3
console.log(codePointLength("bc" ));   // 3
```

此例调用了 `match()` 方法来检查 `text` 中的空白字符与非空白字符（使用 `[\s\S]` 以确保该模式能匹配换行符），所用的正则表达式启用了全局与 Unicode 特性。在匹配至少成功一次的情况下， `result` 变量会是包含匹配结果的数组，因此该数组的长度就是字符串中码点的数量。在 Unicode 中，字符串 `"abc"` 与 `"bc"` 同样包含三个字符，所以数组长度为 3 。

> 这种方法虽然可用，但它不够快，尤其在操作长字符串时。你也可以使用字符串的迭代器（详见第八章）来达到相同目的。一般来说，只要有可能就应尽量减少对码点数量的计算。

判断是否支持 u 标志

既然 `u` 标志是一项语法变更，在不兼容 ES6 的 JS 引擎中试图使用它就会抛出语法错误。最安全的方式是使用如下函数来判断 `u` 标志是否被支持：

```
function hasRegExpU() {
    try {
        var pattern = new RegExp(".", "u");
        return true;
    } catch (ex) {
        return false;
    }
}
```

此函数将 `u` 作为一个参数来调用 `RegExp` 构造器，该语法即便在旧版 JS 引擎中都是有效的，而在 `u` 未被支持的情况下构造器会抛出错误。

> 若你的代码仍需要在旧版 JS 引擎中工作，那么在使用 `u` 标志时应当始终使用 `RegExp` 构造器。这会避免语法错误，并允许你有选择地检测并使用 `u` 标志，而不会导致执行被中断。

字符串的其他改动

JS 字符串的特性总是落后于其它语言，例如，直到 ES5 中字符串才获得了 `trim()` 方法。而 ES6 则继续添加新功能以扩展 JS 解析字符串的能力。

识别子字符串的方法

自从 JS 语言被使用，开发者们就使用 `indexOf()` 方法在字符串内识别子字符串。 ES6 包含了以下三个方法来满足这类需求：

- `includes()` 方法，若给定文本存在于字符串中，会返回 true ，否则返回 false ；
- `startsWith()` 方法，若给定文本出现在字符串起始处，返回 true ，否则返回 false ；
- `endsWith()` 方法，若给定文本出现在字符串结尾处，返回 true ，否则返回 false 。

每个方法都接受两个参数：需要搜索的文本，以及可选的搜索起始位置索引。当提供了第二个参数时， `includes()` 与 `startsWith()` 方法会从该索引位置开始尝试匹配；而 `endsWith()` 方法则会将该位置减去需搜索文本的长度，在计算出的位置尝试匹配。若未提供第二个参数， `includes()` 与 `startsWith()` 方法会从字符串起始处开始查找，而 `endsWith()` 方法的查找则在尾部进行。第二个参数实际上缩小了搜索的范围。以下是使用这些方法的演示：

```
var msg = "Hello world!";

console.log(msg.startsWith("Hello"));       // true
console.log(msg.endsWith("!"));             // true
console.log(msg.includes("o"));             // true

console.log(msg.startsWith("o"));           // false
console.log(msg.endsWith("world!"));        // true
console.log(msg.includes("x"));             // false

console.log(msg.startsWith("o", 4));        // true
console.log(msg.endsWith("o", 8));          // true
console.log(msg.includes("o", 8));          // false
```

前六次调用没有使用第二个参数，因此它们在必要情况下会搜索整个字符串。最后三次调用只检查了字符串的一部分：调用 `msg.startsWith("o", 4)` 从 `msg` 字符串的索引位置 4（即 `"Hello"` 中的 `"o"` ）开始尝试匹配；调用 `msg.endsWith("o", 8)` 会在位置 7 进行匹配尝试，因为需要将参数 `8` 减去字符串 `"o"` 的长度 `1` ；而调用 `msg.includes("o", 8)` 则从索引位置 8 开始尝试匹配，也就是 `"world"` 中的 `"r"` 。

> 原作在此处有错，作者未看出问题的原因是： `"Hello world!"` 这个字符串的第 5 个字符（索引 4 ）与第 8 个字符（索引 7 ）刚好都是字母 `"o"` ， `endsWith()` 的匹配误打误撞地成功了。
>
> 如果修改查找目标，使用 `msg.endsWith("w", 7)` 就能看出明显区别。按原作的说法， `12 - 7 = 5` ，索引 5 （第 6 个字符）是空格 `" "` ，匹配应当失败。但实际上匹配会成功，因为真实的计算是 `7 - 1 = 6` ，索引 6 （第 7 个字符）正是字母 `"w"` 。
>
> 按照 ECMAScript 的规范，当调用 endsWith() 并提供第二个参数时，会用该参数减去欲搜索文本的长度，在计算出的位置对原字符串与欲搜索的文本进行逐个字符比较；若未提供第二个参数，则会使用原字符串的长度去减欲搜索文本的长度。查找过程实际上是正向而非逆向的。
>
> 还是以 `"Hello world!"` 为例，若使用 `msg.endsWith("llo")` ，则实际查找过程是：
>
> 1. `12 - 3 = 9` ，然后在原字符串索引 9 的位置开始匹配。
> 2. 先将原字符串第 10 个字符 `"l"` 与 `"llo"` 的第一个字符 `"l"` 比较，相同，需要继续匹配。
> 3. 再将原字符串第 11 个字符 `"d"` 与 `"llo"` 的第二个字符 `"l"` 比较，不同，匹配失败。
>
> 若改用 `msg.endsWith("llo", 5)` ，则：
>
> 1. `5 - 3 = 2` ，然后在原字符串索引 2 的位置开始匹配。
> 2. 先将原字符串第 3 个字符 `"l"` 与 `"llo"` 的第一个字符 `"l"` 比较，相同，需要继续匹配。
> 3. 再将原字符串第 4 个字符 `"l"` 与 `"llo"` 的第二个字符 `"l"` 比较，依然相同，需要继续匹配。
> 4. 再将原字符串第 5 个字符 `"o"` 与 `"llo"` 的第三个字符 `"o"` 比较，依然相同。
> 5. 全部字符比较完成，匹配成功。

虽然这三个方法使得判断子字符串是否存在变得更容易，但它们的返回结果只是布尔值。若你需要找到确切的匹配位置，则需要使用 `indexOf()` 和 `lastIndexOf()` 。

> 如果向 `startsWith()` 、 `endsWith()` 或 `includes()` 方法传入了正则表达式而不是字符串，会抛出错误。这与 `indexOf()` 以及 `lastIndexOf()` 方法的表现形成了反差，它们会将正则表达式转换为字符串并搜索它。

repeat() 方法

ES6 还为字符串添加了一个 `repeat()` 方法，它接受一个参数作为字符串的重复次数，返回一个将初始字符串重复指定次数的新字符串。例如：

```
console.log("x".repeat(3));         // "xxx"
console.log("hello".repeat(2));     // "hellohello"
console.log("abc".repeat(4));       // "abcabcabcabc"
```

此方法比相同目的的其余方法更加方便，在操纵文本时特别有用，尤其是在需要创建缩进的代码格式化工具中，像这样：

```
// indent 使用了一定数量的空格
var indent = " ".repeat(4),
    indentLevel = 0;

// 每当需要增加缩进
var newIndent = indent.repeat(++indentLevel);
```

第一次调用 `repeat()` 创建了一个包含四个空格的字符串，而 `indentLevel` 变量会持续追踪缩进的级别。此后，仅通过增加 `indentLevel` 的值来调用 repeat() 方法，便可以改变空格数量。

ES6 也为正则表达式的功能进行了一些改进，这些内容不适合纳入特定章节，因此集中在下一节介绍。

正则表达式的其他改动

正则表达式是在 JS 中操作字符串的重要部分之一，与其他方面相似，它在以往的版本中并未有太多变化。不过，为了配合字符串的变更， ES6 也对正则表达式进行了一些改进。

正则表达式 y 标志

在 Firefox 实现了对正则表达式 `y` 标志的专有扩展之后， ES6 随之将其标准化。 `y` 标志影响正则表达式搜索时的粘连（ `sticky` ）属性，它表示在字符串中检索匹配的字符时，应当从正则表达式的 `lastIndex` 属性值的位置开始。如果在该位置没有匹配成功，那么正则表达式将停止检索。参考如下代码以了解其工作机制：

```
var text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello1 "
console.log(stickyResult[0]);   // "hello1 "

pattern.lastIndex = 1;
globalPattern.lastIndex = 1;
stickyPattern.lastIndex = 1;

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello2 "
console.log(stickyResult[0]);   // 错误！ stickyResult 的值为 null
```

此例中有三个正则表达式： `pattern` 中的表达式没有使用任何标志， `globalPattern` 使用了 `g` 标志， `stickyPattern` 则使用了 `y` 标志。对 `console.log()` 的第一次调用，三个正则表达式都分别返回了 `"hello1 "` ，注意此字符串尾部有个空格。

此后，三个模式的 `lastIndex` 属性全部被更改为 1 ，表示三个模式的正则表达式都应当从第二个字符开始尝试匹配。不使用任何标志的正则表达式完全忽略了对于 `lastIndex` 的更改，依然匹配了 `"hello1 "` ；而使用 `g` 标志的正则表达式继续匹配了 `"hello2 "` ，因为它从第二个字符（ `"e"` ）开始，向着字符串尾部方向持续搜索；粘连的正则表达式则在第二个字符处没有匹配成功，因此 `stickyResult` 的值是 `null` 。

一旦匹配操作成功，粘连标志就会将匹配结果之后的那个字符的索引值保存在 `lastIndex` 中；若匹配未成功，那么 `lastIndex` 的值将重置为 0 。全局标志的行为与其相同，如下所示：

```
var text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello1 "
console.log(stickyResult[0]);   // "hello1 "

console.log(pattern.lastIndex);         // 0
console.log(globalPattern.lastIndex);   // 7
console.log(stickyPattern.lastIndex);   // 7

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello2 "
console.log(stickyResult[0]);   // "hello2 "

console.log(pattern.lastIndex);         // 0
console.log(globalPattern.lastIndex);   // 14
console.log(stickyPattern.lastIndex);   // 14
```

对于 `stickyPattern` 和 `globalPattern` 模式变量来说，第一次调用之后 `lastIndex` 的值均被更改为 7 ，而第二次则均被改为 14 。

关于粘连标志，有两个微妙细节需要牢记：

1. 只有调用正则表达式对象上的方法（例如 `exec()` 与 `test()` 方法）， `lastIndex` 属性才会生效。而将正则表达式作为参数传递给字符串上的方法（例如 `match()` ），并不会体现粘连特性。
2. 若使用 `^` 字符来匹配字符串的起始处，粘连的正则表达式只会匹配字符串的起始位置，或在多行模式（ `m` ）下匹配行首。当 `lastIndex` 为 0 时，无论是否使用粘连的正则表达式，对于 `^` 的处理都是一致的；而当 `lastIndex` 在单行模式下不对应于整个字符串起始处，或者当它在多行模式下不对应于行首时，粘连的正则表达式永远不会匹配成功。

和正则表达式其他标志相同，你可以根据一个属性来检测 `y` 标志是否存在。此刻需检查 `sticky` 属性，如下：

```
var pattern = /hello\d/y;

console.log(pattern.sticky);    // true
```

如果粘连标志存在，那么 `sticky` 属性的值会被设为 true ，否则会被设为 false 。 `sticky` 属性由 `y` 标志存在与否决定，是只读的，它的值不能在代码中修改。

与 `u` 标志相似， `y` 标志也是个语法变更，所以在旧版 JS 引擎中它会造成语法错误。你可以用如下方法来检测它是否被支持：

```
function hasRegExpY() {
    try {
        var pattern = new RegExp(".", "y");
        return true;
    } catch (ex) {
        return false;
    }
}
```

此函数类似于对 `u` 标志的检查，在无法使用 `y` 标志来创建正则表达式时会返回 false 。同样，如果使用 `y` 标志的代码可能在旧版 JS 引擎中使用，请确保用 `RegExp` 构造器来定义正则表达式，以避免语法错误。

复制正则表达式

在 ES5 中，你可以将正则表达式传递给 `RegExp` 构造器来复制它，就像这样：

```
var re1 = /ab/i,
    re2 = new RegExp(re1);
```

`re2` 变量只是 `re1` 的一个副本。但如果你向 `RegExp` 构造器传递了第二个参数，即正则表达式的标志，那么该代码就无法工作，正如该范例：

```
var re1 = /ab/i,

    // ES5 中会抛出错误, ES6 中可用
    re2 = new RegExp(re1, "g");
```

如果你在 ES5 环境中运行这段代码，那么你会收到一条错误信息，表示在第一个参数已经是正则表达式的情况下，不能再使用第二个参数。 ES6 则修改了这个行为，允许使用第二个参数，并且让它覆盖第一个参数中的标志。例如：

```
var re1 = /ab/i,

    // ES5 中会抛出错误, ES6 中可用
    re2 = new RegExp(re1, "g");

console.log(re1.toString());            // "/ab/i"
console.log(re2.toString());            // "/ab/g"

console.log(re1.test("ab"));            // true
console.log(re2.test("ab"));            // true

console.log(re1.test("AB"));            // true
console.log(re2.test("AB"));            // false
```

此代码中的 `re1` 带有忽略大小写的 `i` 标志，而 `re2` 则只带有全局的 `g` 标志。 `RegExp` 构造器复制了 `re1` 的模式并用 `g` 标志替换了 `i` 标志。如果没有第二个参数， `re2` 就会拥有与 `re1` 相同的标志。

flags属性

除了新增一个标志，并且扩充了使用标志的方式， ES6 还新增了一个与标志关联的属性。在 ES5 中，你可以使用 `source` 属性来获取正则表达式的文本，但若想获取标志字符串，你必须解析 `toString()` 方法的输出，就像下面展示的那样：

```
function getFlags(re) {
    var text = re.toString();
    return text.substring(text.lastIndexOf("/") + 1, text.length);
}

// toString() 的输出为 "/ab/g"
var re = /ab/g;

console.log(getFlags(re));          // "g"
```

此处将正则表达式转换为一个字符串，并返回了最后一个 `/` 之后的字符，这些字符即为该正则表达式所用的标志。

ES6 新增了 `flags` 属性用于配合 `source` 属性，让标志的获取变得更容易。这两个属性均为只有 getter 的原型访问器属性，因此都是只读的。 `flags` 属性让检查正则表达式更容易，有助于调试与继承方面的工作。

ES6 后期加入的 `flags` 属性，会返回由正则表达式中所有标志组成的字符串。例如：

```
var re = /ab/g;

console.log(re.source);     // "ab"
console.log(re.flags);      // "g"
```

本例查找了 `re` 的所有标志并将其打印到控制台，所用的代码量大大少于 `toString()` 方式。同时使用 `source` 和 `flags` 允许你直接提取正则表达式的组成部分，而不必将正则表达式转换为字符串。

关于字符串与正则表达式，本章介绍过的改进已绝对强大，然而 ES6 在字符串方面还有更大的扩展，它引入了一种新的字面量形式让字符串的使用更加灵活。

模板字面量

JS 的字符串相对其他语言来说功能总是有限的。例如，本章介绍过的字符串方法在 ES6 之前都缺失，而字符串的拼接功能则极尽简陋。为了让开发者能够解决复杂的问题， ES6 的模板字面量（ **template literal** ）提供了创建领域专用语言的语法，与 ES5 及更早版本的解决方案相比，处理内容可以更安全。领域专用语言（ domain-specific language ， DSL ）是被设计用于特定有限目的的编程语言，与 JavaScript 这样通用目的的语言相反。 ECMAScript wiki 在 template literal strawman 上提供了如下描述：

> 本方案通过语法糖扩展了 ECMAScript 的语法，允许语言库提供 DSL 以便生成、查询并操纵来自于其它语言的内容，并且能够对 XSS 、 SQL 注入等注入攻击免疫，或具有抗性。

不过实际上，JS 语言直到 ES5 依然完全缺失如下功能，模板字面量是 ES6 针对这些功能的回应：

- 多行字符串：针对多行字符串的正式概念；
- 基本的字符串格式化：使用已存在的变量值，对字符串进行部分替换的能力；
- **HTML** 转义：能转换字符串以便将其安全插入到 HTML 中的能力。

模板字面量以一种全新方式解决了这些问题，而并未给 JS 已有的字符串添加额外功能。

基本语法

模板字面量的最简单语法，是使用反引号（ `` ` `` ）来包裹普通字符串，而不是用双引号或单引号。参考以下例子：

```
let message = `Hello world!`;

console.log(message);             // "Hello world!"
console.log(typeof message);      // "string"
console.log(message.length);      // 12
```

此代码说明了 `message` 变量包含的是一个普通的 JS 字符串。模板字面量语法被用于创建一个字符串值，并被赋值给了 `message` 变量。

若你想在字符串中包含反引号，只需使用反斜杠（ `\` ）转义即可，就像下面这个版本的 `message` 变量：

```
let message = `\`Hello\` world!`;

console.log(message);             // "`Hello` world!"
console.log(typeof message);      // "string"
console.log(message.length);      // 14
```

在模板字面量中无需对双引号或单引号进行转义。

多行字符串

JS 开发者从语言诞生起就一直寻找一种能创建多行字符串的方法。但在使用双引号或单引号时，整个字符串只能放在同一行。

ES6 之前的权宜之计

感谢存在已久的一个语法 bug ， JS 的确有用于创建多行字符串的权宜之计，在换行之前使用反斜线（ `\` ）即可。此处有个范例：

```
var message = "Multiline \
string";

console.log(message);       // "Multiline string"
```

`message` 字符串打印输出时不会有换行，因为反斜线被视为行的续延符号，而非新行的符号。为了在输出中显示换行，你需要手动加入它：

```
var message = "Multiline \n\
string";

console.log(message);       // "Multiline
                            //  string"
```

在所有主流的 JS 引擎中，此代码都会输出两行，但是该行为被认定为一个 bug ，并被许多开发者建议规避。

ES6 之前创建多行字符串的其他尝试，一般都基于数组或字符串的拼接，就像这样：

```
var message = [
    "Multiline ",
    "string"
].join("\n");

let message = "Multiline \n" +
    "string";
```

关于 JS 缺失的多行字符串功能，开发者的所有解决方法都不够完美。

多行字符串的简单解决方法

ES6 的模板字面量让多行字符串更易创建，因为它不需要特殊的语法。只需在想要的位置包含换行即可，它会包含在结果字符串中。例如：

```
let message = `Multiline
string`;

console.log(message);           // "Multiline
                                //  string"
console.log(message.length);    // 16
```

反引号之内的所有空白符都是字符串的一部分，因此需要留意缩进。例如：

```
let message = `Multiline
               string`;

console.log(message);           // "Multiline
                                //                 string"
console.log(message.length);    // 31
```

此代码中，模板字面量第二行前面的所有空白符都被视为字符串自身的一部分。如果让多行文本保持合适的缩进对你来说很重要，请考虑将多行模板字面量的第一行空置，并在此后进行缩进，如下所示：

```
let html = `
<div>
    <h1>Title</h1>
</div>`.trim();
```

此代码从第一行开始创建模板字面量，但有效文本从第二行才开始。 HTML 标签的缩进增强了可读性，之后再调用 trim() 方法移除了起始的空行。

> 如果你喜欢的话，也可以在模板字面量中使用 `\n` 来指示换行的插入位置：
>
> ```
> let message = `Multiline\nstring`;
>
> console.log(message);           // "Multiline
> //  string"
> console.log(message.length);    // 16
> ```

产生替换位

模板字面量目前看上去仅仅是普通 JS 字符串的升级版，但二者之间真正的区别在于前者的“替换位”。替换位允许你将任何有效的 JS 表达式嵌入到模板字面量中，并将其结果输出为字符串的一部分。

替换位由起始的 `${` 与结束的 `}` 来界定，之间允许放入任意的 JS 表达式。最简单的替换位允许你将本地变量直接嵌入到结果字符串中，例如：

```
let name = "Nicholas",
    message = `Hello, ${name}.`;

console.log(message);       // "Hello, Nicholas."
```

替换位 `${name}` 会访问本地变量 `name` ，并将其值插入到 `message` 字符串中。 `message` 变量会立即持有该变量的值。

> 模板字面量能访问到作用域中任意的可访问变量。试图使用未定义的变量会抛出错误，无论是否为严格模式。

既然替换位是 JS 表达式，那么可替换的就不仅仅是简单的变量名。你可以轻松嵌入计算、函数调用等等。例如：

```
let count = 10,
    price = 0.25,
    message = `${count} items cost $${(count * price).toFixed(2)}.`;

console.log(message);       // "10 items cost $2.50."
```

此代码在模板字面量的一部分执行了一次计算， `count` 与 `price` 变量相乘，再使用 `.toFixed()` 方法将结果格式化为两位小数。而在第二个替换位之前的美元符号被照常输出，因为没有左花括号紧随其后。

模板字面量本身也是 JS 表达式，意味着你可以将模板字面量嵌入到另一个模板字面量内部，如同下例：

```
let name = "Nicholas",
    message = `Hello, ${
        `my name is ${ name }`
    }.`;

console.log(message);        // "Hello, my name is Nicholas."
```

此例在第一个模板字面量内又嵌入了另一个。在首个 `${` 之后使用了另一个模板字面量，第二个 `${` 标示了嵌入到内层模板字面量的表达式的开始，该表达式为 `name` 变量，其值被插入结果。

标签化模板

现在你已了解到，模板字面量在无须拼接操作的情况下，是如何创建多行字符串，以及如何将值插入字符串的。不过模板字面量真实威力其实来源于标签化模板。模板标签（ **template tag** ）能对模板字面量进行转换并返回最终的字符串值，标签在模板的起始处被指定，即在第一个 `` ` `` 之前，如下所示：

```
let message = tag`Hello world`;
```

在本例中， `tag` 即为应用到 `` `Hello world` `` 模板字面量上的模板标签。

定义标签

一个标签（ **tag** ）仅是单纯的函数，它被调用时接收需要处理的模板字面量数据。标签所接收的数据被划分为独立片段，并且应当将它们组合起来以产生结果。第一个参数是个数组，包含被 JS 解释过的字面量字符串，随后的参数是每个替换位的解释值。

标签函数的参数一般定义为剩余参数形式，以便更方便处理数据，如下：

```
function tag(literals, ...substitutions) {
    // 返回一个字符串
}
```

为了更好理解传递给标签的参数是什么，可研究下例：

```
let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $${(count * price).toFixed(2)}.`;
```

若存在一个名为 `passthru()` 的函数，该函数将会接收到三个参数。首先是一个 `literals` 数组，包含如下元素：

- 在首个替换位之前的空字符串（ `""` ）；
- 首个替换位与第二个替换位之间的字符串（ `" items cost $"` ）；
- 第二个替换位之后的字符串（ `"."` ）。

接下来的参数会是 `10` ，也就是 `count` 变量的解释值，它也会成为 `substitutions` 数组的第一个元素。最后一个参数则会是 `"2.50"` ，即 `(count * price).toFixed(2)` 的解释值，并且会是 `substitutions` 数组的第二个元素。

需要注意 `literals` 的第一个元素是空字符串，以确保 `literals[0]` 总是字符串的起始部分，正如 `literals[literals.length - 1]` 总是字符串的结尾部分。同时，替换位的元素数量也总是比字面量元素少 1 ，意味着表达式 `substitutions.length === literals.length - 1` 的值总是 true 。

使用这种模式，可以交替使用 `literals` 与 `substitutions` 数组来生成一个结果字符串：以 `literals` 中的首个元素开始，后面紧跟着 `substitutions` 中的首个元素，如此反复，直到结果字符串被生成完毕。你可以像下例这样交替使用两个数组中的值来模拟模板字面量的默认行为：

```
function passthru(literals, ...substitutions) {
    let result = "";

    // 仅使用 substitution 的元素数量来进行循环
    for (let i = 0; i < substitutions.length; i++) {
        result += literals[i];
        result += substitutions[i];
    }

    // 添加最后一个字面量
    result += literals[literals.length - 1];

    return result;
}

let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $${(count * price).toFixed(2)}.`;

console.log(message);       // "10 items cost $2.50."
```

本例定义了 `passthru` 标签，所执行的转换操作与模板字面量的默认行为相同。唯一的诀窍是在循环中使用 `substituions.length` 而不是 `literals.length` 来避免 `substituions` 数组的越界。它能工作是由于 ES6 对 `literals` 和 `substituions` 的良好定义。

> `substituions` 中包含的值不必是字符串。若表达式的计算结果为数字（就像上例），那么该数值也会被传入。决定这些值在结果中如何输出是标签的职责。

使用模板字面量中的原始值

模板标签也能访问字符串的原始信息，主要指的是可以访问字符在转义之前的形式。获取原始字符串值的最简单方式是使用内置的 `String.raw()` 标签。例如：

```
let message1 = `Multiline\nstring`,
    message2 = String.raw`Multiline\nstring`;

console.log(message1);          // "Multiline
                                //  string"
console.log(message2);          // "Multiline\\nstring"
```

此代码中， `message1` 中的 `\n` 被解释为一个换行，而 `message2` 中的 `\n` 返回了它的原始形式 `"\\n"` （反斜线与 `n` 字符）。在必要时，如此提取原始字符串信息可以进行更复杂的处理。

字符串的原始信息同样会被传递给模板标签。标签函数的第一个参数为包含额外 `raw` 属性的数组，而 `raw` 属性则是含有与每个字面量值等价的原始值的数组。例如， `literals[0]` 的值总是等价于 `literals.raw[0]` 的值，后者包含字符串的原始信息。知道这些之后，你可以用如下代码来模拟 `String.raw()` ：

```
function raw(literals, ...substitutions) {
    let result = "";

    // 仅使用 substitution 的元素数量来进行循环
    for (let i = 0; i < substitutions.length; i++) {
        result += literals.raw[i];      // 改为使用原始值
        result += substitutions[i];
    }

    // 添加最后一个字面量
    result += literals.raw[literals.length - 1];

    return result;
}

let message = raw`Multiline\nstring`;

console.log(message);           // "Multiline\\nstring"
console.log(message.length);    // 17
```

这里使用 `literals.raw` 而非 `literals` 来输出结果字符串。这意味着任何转义字符（包括 Unicode 码点的转义）都会以原始的形式返回。若想在输出的字符串中包含转义字符，原始字符串会很有帮助。例如，若想要生成包含代码的文档，那么应当输出实际的原始代码而非

转义后的结果。

总结

具备了完整的 Unicode 支持，JS 就能以合理的方式处理 UTF-16 字符。通过 `codePointAt()` 与 `String.fromCodePoint()` 在码点和字符之间转换，是字符串操作能力的一大进步。正则表达式新增的 `u` 标志让直接操作码点而不是 16 位字符变为可能，而 `normalize()` 方法则允许进行更准确的字符串比较。

ES6 也添加了操作字符串的新方法，允许你更容易识别子字符串，而不用在意它在父字符串中的位置。正则表达式同样引入了许多功能。

模板字面量是 ES6 的一项重要补充，允许你创建领域专用语言（ DSL ），让字符串的创建更容易。能将变量直接嵌入到模板字面量中，意味着开发者在组合长字符串与变量时，有了一种比字符串拼接更为安全的工具。

内置的多行字符串支持，超越了普通 JS 字符串的能力，这让模板字面量的升级更加有用。尽管在模板字面量中允许直接使用换行，你依然可以使用 `\n` 或其它字符转义序列。

模板标签是创建 DSL 最重要的部分。标签是接收模板字面量片段作为参数的函数，你可以使用它们来返回合适的字符串。传入参数包括了字面量、等价的原始值以及替换位的值，标签使用这些信息片段来决定输出。

第三章 函数

函数在任何编程语言中都是非常重要的一部分，而从 JS 诞生起，一直到 ES6 之前，函数都未有较大的变化。这积压了诸多问题及细微行为差异，由此容易诱发错误，并且经常需要用大量代码来实现非常基本的功能。

ES6 的函数考虑了 JS 开发者多年的抱怨与诉求，大踏步前进，在 ES5 函数基础上实现了不少增量改进，让 JS 更加强大，同时编程错误也更少。

带参数默认值的函数

JS 函数的独特之处是可以接受任意数量的参数，而无视函数声明处的形参数量。这让你定义的函数可以使用不同的参数数量来调用，而未提供的参数经常会使用默认值来代替。本章将介绍默认的参数值在 ES6 前后是如何实现的，顺带介绍的内容还有： `arguments` 对象的一些重要信息，将表达式作为参数使用，以及另一种形式的 TDZ 。

在 ES5 中模拟参数默认值

在 ES5 或更早的版本中，你或许会使用以下模式来创建带有参数默认值的函数：

```
function makeRequest(url, timeout, callback) {

    timeout = timeout || 2000;
    callback = callback || function() {};

    // 函数的剩余部分

}
```

在本例中， `timeout` 与 `callback` 实际上都是可选参数，因为他们都会在参数未被提供的情况下使用默认值。逻辑或运算符（ `||` ）在左侧的值为假值（ falsy ）的情况下总会返回右侧的操作数。由于函数的具名参数在未被明确提供时会是 `undefined` ，逻辑或运算符就经常被用来给缺失的参数提供默认值。不过此方法有个瑕疵，此处的 `timeout` 的有效值实际上有可能是 `0` ，但因为 `0` 是假值，就会导致 `timeout` 的值在这种情况下被替换为 `2000` 。

在这种情况下，更安全的选择是使用 `typeof` 来检测参数的类型，正如下例：

```
function makeRequest(url, timeout, callback) {

    timeout = (typeof timeout !== "undefined") ? timeout : 2000;
    callback = (typeof callback !== "undefined") ? callback : function() {};

    // 函数的剩余部分

}
```

尽管这种方式更安全，却为实现一个基本需求而书写了过多的代码。它代表了一种常见模式，充斥在各种流行的 JS 库中。

ES6 中的参数默认值

ES6 能更容易地为参数提供默认值，它使用了初始化形式，以便在参数未被正式传递进来时使用。例如：

```
function makeRequest(url, timeout = 2000, callback = function() {}) {

    // 函数的剩余部分

}
```

此函数只要求始终提供第一个参数。其余两个参数则都有默认值，这就无需再添加更多代码来检查缺失的参数值，让函数体更为小巧。

若完整使用三个参数来调用 `makeRequest()` ，那么默认值将不会被用到，例如：

```
// 使用默认的 timeout 与 callback
makeRequest("/foo");

// 使用默认的 callback
makeRequest("/foo", 500);

// 不使用默认值
makeRequest("/foo", 500, function(body) {
    doSomething(body);
});
```

ES6 会认为 `url` 参数是必须的，因此三次调用 `makeRequest()` 都必须传入 `"/foo"` 。拥有默认值的两个参数则都被认为是可选的。

在函数声明中能指定任意一个参数的默认值，即使之后还存在未指定默认值的参数。例如，下面这样是可行的：

```
function makeRequest(url, timeout = 2000, callback) {

    // 函数的剩余部分

}
```

在本例中，只有在未传递第二个参数、或明确将第二个参数值指定为 `undefined` 时，`timeout` 的默认值才会被使用，例如：

```
// 使用默认的 timeout
makeRequest("/foo", undefined, function(body) {
    doSomething(body);
});

// 使用默认的 timeout
makeRequest("/foo");

// 不使用默认值
makeRequest("/foo", null, function(body) {
    doSomething(body);
});
```

在这个例子中， `null` 值被认为是有效参数，意味着对于 `makeRequest()` 的第三次调用并不会使用 `timeout` 的默认值。

参数默认值如何影响 arguments 对象

需要记住的是， `arguments` 对象会在使用参数默认值时有不同的表现。在 ES5 的非严格模式下， `arguments` 对象会反映出具名参数的变化。以下代码说明了该工作机制：

```
function mixArgs(first, second) {
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d";
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a", "b");
```

输出：

```
true
true
true
true
```

在非严格模式下， `arguments` 对象总是会被更新，以反映出具名参数的变化。因此当 `first` 与 `second` 变量被赋予新值时， `arguments[0]` 与 `arguments[1]` 也就相应被更新，让这里所有的 `===` 严格比较的结果都为 `true` 。

然而在 ES5 的严格模式下，消除了关于 `arguments` 对象的这种混乱情况，它不再反映出具名参数的变化。在严格模式下重新使用上例中的函数：

```
function mixArgs(first, second) {
    "use strict";

    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a", "b");
```

调用 `mixArgs()` 则输出：

```
true
true
false
false
```

这一次更改 `first` 与 `second` 就不会再影响 `arguments` 对象，因此输出结果符合通常的期望。

然而在使用 ES6 参数默认值的函数中，无论函数是否明确运行在严格模式下， `arguments` 对象的表现总是会与 ES5 的严格模式一致，参数默认值的存在触发了 `arguments` 对象与具名参数的分离。这是个细微但重要的细节，表示 `arguments` 对象的使用方式发生了变化。研究如下代码：

```
// 非严格模式
function mixArgs(first, second = "b") {
    console.log(arguments.length);
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a");
```

输出：

```
1
true
false
false
false
```

本例中 `arguments.length` 的值为 `1` ，因为只给 `mixArgs()` 传递了一个参数。这也意味着 `arguments[1]` 的值是 `undefined` ，符合将单个参数传递给函数时的预期；这同时意味着 `first` 与 `arguments[0]` 是相等的。无论是否在严格模式下，改变 `first` 和 `second` 的值不会对 `arguments` 对象造成影响，所以 `arguments` 对象始终能映射出初始调用状态。

参数默认值表达式

参数默认值最有趣的特性，或许就是默认值并不非得是基本类型值。例如，你可以执行一个函数来产生参数的默认值，就像这样：

```
function getValue() {
    return 5;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 6
```

此处若未提供第二个参数， `getValue()` 函数就会被调用以获取正确的默认值。需要注意的是，仅在调用 `add()` 函数而未提供第二个参数时， `getValue()` 函数才会被调用，而在 `add()` 的函数声明初次被解析时并不会进行调用。这意味着 `getValue()` 函数若被写为可变的，则默认参数获取的值有可能也会变化，例如：

```
let value = 5;

function getValue() {
    return value++;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 6
console.log(add(1));        // 7
```

本例中 `value` 的初始值是 5 ，并且会随着对 `getValue()` 的每次调用而递增。首次调用 `add(1)` 返回的值为 6 ，再次调用时 `value` 的值已被增加，于是返回了 7 。由于 `second` 参数的默认值总是在 `add()` 函数被调用的情况下才被计算，因此该参数的值随时都会被改变。

> 将函数调用作为参数的默认值时需要小心，如果你遗漏了括号，例如在上面例子中使用 `second = getValue` ，你就传递了对于该函数的一个引用，而没有传递调用该函数的结果。

这种行为引出了另一种有趣的能力：可以将前面的参数作为后面参数的默认值，此处有个例子：

```
function add(first, second = first) {
    return first + second;
}

console.log(add(1, 1));      // 2
console.log(add(1));         // 2
```

此代码中 `first` 为 `second` 参数提供了默认值，意味着只传入一个参数会让两个参数获得相同的值，因此 `add(1, 1)` 与 `add(1)` 同样返回了 2 。再进一步，你还能将 `first` 作为参数传递给一个函数来产生 `second` 参数的值，正如下例：

```
function getValue(value) {
    return value + 5;
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));      // 2
console.log(add(1));         // 7
```

此例将 `second` 的值设为等于 `getValue(first)` 函数的返回值，因此 `add(1)` 会返回 7 （ 1 + 6 ），而 `add(1, 1)` 仍然返回 2 。

引用其他参数来为参数进行默认赋值时，仅允许引用前方的参数，因此前面的参数不能向后访问，例如：

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));          // 2
console.log(add(undefined, 1)); // 抛出错误
```

调用 `add(undefined, 1)` 发生了错误，这是由于 `second` 在 `first` 之后定义，因此不能将其作为后者的默认值。为了理解这种情况的成因，需着重回顾“暂时性死区”概念。

参数默认值的暂时性死区

第一章介绍了 `let` 与 `const` 的暂时性死区（ TDZ ），而参数默认值同样具有暂时性死区。与 `let` 声明相似，函数每个参数都会创建一个新的标识符绑定，它在初始化之前不允许被访问，否则会抛出错误。参数初始化会在函数被调用时进行，无论是给参数传递了一个值、还是使用了参数的默认值。

为了探索参数默认值中的暂时性死区，可再次研究“参数默认值表达式”中的例子：

```
function getValue(value) {
    return value + 5;
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 7
```

调用 `add(1, 1)` 和 `add(1)` 事实上执行了以下代码来创建 `first` 与 `second` 的参数值：

```
// JS 调用 add(1, 1) 可表示为
let first = 1;
let second = 1;

// JS 调用 add(1) 可表示为
let first = 1;
let second = getValue(first);
```

当函数 `add()` 第一次执行时， `first` 与 `second` 的绑定被加入了特定参数的暂时性死区（类似于 `let` 声明的行为）。因此 `second` 可以使用 `first` 来初始化，因为此处 `first` 已完成了初始化，但反之则不行。现在再研究以下重写过的 `add()` 函数：

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));         // 2
console.log(add(undefined, 1)); // 抛出错误
```

本例中调用 `add(1, 1)` 与 `add(undefined, 1)` 对应着以下的后台代码：

```
// JS 调用 add(1, 1) 可表示为
let first = 1;
let second = 1;

// JS 调用 add(1) 可表示为
let first = second;
let second = 1;
```

本例中调用 `add(undefined, 1)` 抛出了错误，是因为在对 `first` 进行初始化时， `second` 尚未被初始化。此处的 `second` 位于暂时性死区内，对 `second` 的引用就抛出了错误，正如第一章讨论过的 `let` 绑定的行为。

> 函数参数拥有各自的作用域和暂时性死区，与函数体的作用域相分离，这意味着参数的默认值不允许访问在函数体内部声明的任意变量。

使用不具名参数

到目前为止，本章的例子只涵盖了在函数定义中的已被命名的参数。然而 JS 的函数并不强求参数的数量要等于已定义具名参数的数量，实际所传递的参数允许少于或多于正式指定的参数。参数的默认值让函数在接收更少参数时的行为更清晰，而 ES6 试图让相反情况的问题也被更好地解决。

ES5 中的不具名参数

JS 早就提供了 `arguments` 对象用于查看传递给函数的所有参数，这样就不必分别指定每个参数。虽然查看 `arguments` 对象在大多数情况下都工作正常，但操作它有时仍然较为麻烦。例如，参考以下查看 `arguments` 对象的代码：

```
function pick(object) {
    let result = Object.create(null);

    // 从第二个参数开始处理
    for (let i = 1, len = arguments.length; i < len; i++) {
        result[arguments[i]] = object[arguments[i]];
    }

    return result;
}

let book = {
    title: "Understanding ES6",
    author: "Nicholas C. Zakas",
    year: 2015
};

let bookData = pick(book, "author", "year");

console.log(bookData.author);   // "Nicholas C. Zakas"
console.log(bookData.year);     // 2015
```

此函数模拟了 **Underscore.js** 代码库的 `pick()` 方法，能够返回包含原有对象特定属性的子集副本。本例中只为函数定义了一个期望参数，也就是拷贝属性的来源对象，除此之外传递的所有参数则都是需要拷贝的属性的名称。

这个 `pick()` 函数有两点需要注意。首先，完全看不出该函数具备处理多个参数的能力，就算为其再多定义几个参数，但依然不足以标明它能处理任意数量的参数。其次，由于第一个参数被命名并被直接使用，当你寻找需要复制的属性时，就必须从 `arguments` 对象索引位置 1 而非位置 0 开始处理。要记住使用 `arguments` 的适当索引值或许不困难，但毕竟多了一件挂心之事。

ES6 引入了剩余参数以便解决此问题。

剩余参数

剩余参数（ **rest parameter** ）由三个点（ `...` ）与一个紧跟着的具名参数指定，它是包含传递给函数的其余参数的一个数组，由此得名“剩余”。例如， `pick()` 函数可以像下面这样用剩余参数来重写：

```
function pick(object, ...keys) {
    let result = Object.create(null);

    for (let i = 0, len = keys.length; i < len; i++) {
        result[keys[i]] = object[keys[i]];
    }

    return result;
}
```

在此版本的函数中， `keys` 是一个剩余参数，包含所有在 `object` 之后的参数，这与囊括所有参数的 `arguments` 不同，后者连第一个参数都会包含。这意味着你无需有所顾虑，可以对 `keys` 从头到尾进行迭代。作为一个额外的收益，通过观察该函数声明便能判明它具有处理任意数量参数的能力。

> 函数的 `length` 属性用于指示具名参数的数量，而剩余参数对其毫无影响。此例中 `pick()` 函数的 `length` 属性值是 1 ，因为只有 `object` 参数被用于计算该值。
>
> 译注：这种说法并不严谨。若函数使用了默认参数，则 **`length`** 属性不包含使用默认值的参数，并且它只能指示出第一个默认参数之前的具名参数数量。例如对于 **`function example(first, second = 'woo', third) {}`** 函数声明来说， **`length`** 的值是 **`1`** 而非 **`2`** ，尽管这里有两个无默认值的具名参数。

剩余参数的限制条件

剩余参数受到两点限制。一是函数只能有一个剩余参数，并且它必须被放在最后。例如，如下代码是无法工作的：

```
// 语法错误：不能在剩余参数后使用具名参数
function pick(object, ...keys, last) {
    let result = Object.create(null);

    for (let i = 0, len = keys.length; i < len; i++) {
        result[keys[i]] = object[keys[i]];
    }

    return result;
}
```

此处的 `last` 跟在了剩余参数 `keys` 后面，这会导致一个语法错误。

第二个限制是剩余参数不能在对象字面量的 setter 属性中使用，这意味着如下代码同样会导致语法错误：

```
let object = {

    // 语法错误：不能在 setter 中使用剩余参数
    set name(...value) {
        // 一些操作
    }
};
```

存在此限制的原因是：对象字面量的 setter 被限定只能使用单个参数；而剩余参数按照定义是不限制参数数量的，因此它在此处不被许可。

剩余参数如何影响 **arguments** 对象

设计剩余参数是为了替代 ES 中的 `arguments` 对象。ES4 曾经移除了 `arguments` 并添加了剩余参数，以便允许向函数传入不限数量的参数。尽管 ES4 规范被废弃，但这个想法被保持下来，并在 ES6 中被重新引入，不过 `arguments` 仍得以保留。

`arguments` 对象在函数被调用时反映了传入的参数，与剩余参数能协同工作，就像如下程序所演示的：

```
function checkArgs(...args) {
    console.log(args.length);
    console.log(arguments.length);
    console.log(args[0], arguments[0]);
    console.log(args[1], arguments[1]);
}

checkArgs("a", "b");
```

调用 `checkArgs()` 输出了：

```
2
2
a a
b b
```

`arguments` 对象总能正确反映被传入函数的参数，而无视剩余参数的使用。

对剩余参数需要了解的内容已介绍完毕，你可以将其投入使用了。

函数构造器的增强

`Function` 构造器允许你动态创建一个新函数，但在 JS 中并不常用。传给该构造器的参数都是字符串，它们就是目标函数的参数与函数体，此处有个范例：

```
var add = new Function("first", "second", "return first + second");

console.log(add(1, 1));     // 2
```

ES6 增强了 `Function` 构造器的能力，允许使用默认参数以及剩余参数。对于默认参数来说，你只需为参数名称添加等于符号以及默认值，正如下例：

```
var add = new Function("first", "second = first",
        "return first + second");

console.log(add(1, 1));     // 2
console.log(add(1));        // 2
```

在此例中，当只传递了一个参数时， `first` 的值会被赋给 `second` 参数，此处的语法与不使用 `Function` 的函数声明一致。

而对剩余参数来说，只需在最后一个参数前添加 `...` 即可，就像这样：

```
var pickFirst = new Function("...args", "return args[0]");

console.log(pickFirst(1, 2));   // 1
```

此代码创建了一个仅使用剩余参数的函数，让其返回所传入的第一个参数。

为 `Function` 构造器添加默认参数与剩余参数，确保了它拥有与函数声明形式相同的所有能力。

扩展运算符

与剩余参数关联最密切的就是扩展运算符。剩余参数允许你把多个独立的参数合并到一个数组中；而扩展运算符则允许将一个数组分割，并将各个项作为分离的参数传给函数。考虑一下 `Math.max()` 方法，它接受任意数量的参数，并会返回其中的最大值。此处有个简单用例：

```
let value1 = 25,
    value2 = 50;

console.log(Math.max(value1, value2));      // 50
```

若像本例这样仅处理两个值，那么仅需将这两个值直接传入 `Math.max()` ，较大的那个就会成为返回值。但如何处理数组中的多个值呢？ `Math.max()` 方法并不允许你传入一个数组，因此在 ES5 或更早版本中，要么自行搜索整个数组，要么像下面这样使用 `apply()` 方法：

```
let values = [25, 50, 75, 100]

console.log(Math.max.apply(Math, values));  // 100
```

该解决方案是可行的，但这样使用 `apply()` 会稍微令人疑惑，它使用额外语法而混淆了代码的真实意图。

ES6 的扩展运算符令这种情况变得简单。无须调用 `apply()` ，你可以在该数组前添加 `...` 并直接将其传递给 `Math.max()` ，就像使用剩余参数那样。 JS 引擎将会将该数组分割为独立参数并把它们传递进去：

```
let values = [25, 50, 75, 100]

// 等价于 console.log(Math.max(25, 50, 75, 100));
console.log(Math.max(...values));           // 100
```

如此调用 `Math.max()` 更接近传统形式，并避免了为一个简单数学操作使用复杂的 `this` 绑定（即在上个例子中提供给 `Math.max.apply()` 的第一个参数）。

你可以将扩展运算符与其他参数混用。假设你想让 `Math.max()` 返回的最小值为 0 ，以忽略数组内的负值，可以将参数 0 单独传入，并继续为其他参数使用扩展运算符，正如下例：

```
let values = [-25, -50, -75, -100]

console.log(Math.max(...values, 0));        // 0
```

本例中传给 `Math.max()` 的最后一个参数是 `0` ，它跟在使用扩展运算符的其他参数之后。

用扩展运算符传递参数，让函数更容易用数组作为参数，在大部分场景中，扩展运算符都是 `apply()` 方法的合适替代品。

关于 ES6 中的默认参数与剩余参数，除了你至今看到的用法之外，还可以在 `Function` 构造器中使用它们。

ES6 的名称属性

定义函数有各种各样的方式，在 JS 中识别函数因此变得很有挑战性。此外，匿名函数表达式的流行使得调试有点困难，经常导致堆栈跟踪难以被阅读与解释。为此， ES6 给所有函数添加了 `name` 属性。

选择合适的名称

ES6 中所有函数都有适当的 `name` 属性值。为了理解其实际运作，请看下例——它展示了一个函数与一个函数表达式，并将二者的 `name` 属性都打印出来：

```
function doSomething() {
    // ...
}

var doAnotherThing = function() {
    // ...
};

console.log(doSomething.name);          // "doSomething"
console.log(doAnotherThing.name);       // "doAnotherThing"
```

在此代码中，由于 `doSomething()` 是一个函数声明，它就拥有一个值为 `"doSomething"` 的 `name` 属性。而匿名函数表达式 `doAnotherThing()` 的 `name` 属性值则是 `"doAnotherThing"`，因为这是该函数所赋值的变量的名称。

> 译注：匿名函数的名称属性在 FireFox 与 Edge 中仍然不被支持，值为空字符串，而 Chrome 直到 51.0 版本才提供了该特性。

名称属性的特殊情况

虽然函数声明与函数表达式的名称易于确定，但 ES6 还做了更多规定以确保所有函数都拥有合适的名称。为了表明这点，请参考如下程序：

```
var doSomething = function doSomethingElse() {
    // ...
};

var person = {
    get firstName() {
        return "Nicholas"
    },
    sayName: function() {
        console.log(this.name);
    }
}

console.log(doSomething.name);      // "doSomethingElse"
console.log(person.sayName.name);   // "sayName"

var descriptor = Object.getOwnPropertyDescriptor(person, "firstName");
console.log(descriptor.get.name); // "get firstName"
```

本例中的 `doSomething.name` 的值是 `"doSomethingElse"` ，因为该函数表达式自己拥有一个名称，并且此名称的优先级要高于赋值目标的变量名。 `person.sayName()` 的 `name` 属性值是 `"sayName"` ，正如对象字面量指定的那样。类似的， `person.firstName` 实际是个 getter 函数，因此它的名称是 `"get firstName"` ，以标明它的特征；同样， setter 函数也会带有 `"set"` 的前缀（ getter 与 setter 函数都必须用 `Object.getOwnPropertyDescriptor()` 来检索）。

函数名称还有另外两个特殊情况。使用 `bind()` 创建的函数会在名称属性值之前带有 `"bound"` 前缀；而使用 `Function` 构造器创建的函数，其名称属性为 `"anonymous"` ，正如此例：

```
var doSomething = function() {
    // ...
};

console.log(doSomething.bind().name);   // "bound doSomething"

console.log((new Function()).name);     // "anonymous"
```

绑定产生的函数拥有原函数的名称，并总会附带 `"bound"` 前缀，因此 `doSomething()` 函数的绑定版本的名称为 `"bound doSomething"` 。

> 需要注意的是，函数的 `name` 属性值未必会关联到同名变量。 `name` 属性是为了在调试时获得有用的相关信息，所以不能用 `name` 属性值去获取对函数的引用。

明确函数的双重用途

在 ES5 以及更早版本中，函数根据是否使用 `new` 去调用而有双重用途。当使用 `new` 时，函数内部的 `this` 是一个新对象，并作为函数的返回值，如下例所示：

```
function Person(name) {
    this.name = name;
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");

console.log(person);        // "[Object object]"
console.log(notAPerson);    // "undefined"
```

当调用 `Person()` 来创建 `notAPerson` 时，未使用 `new` ，输出了 `undefined` ，并且在非严格模式下给全局对象添加了 `name` 属性。 `Person` 首字母大写是指示其应当使用 `new` 来调用的唯一标识，这在 JS 编程中是个惯例。函数双重角色的混乱情况在 ES6 中发生了一些改变。

JS 为函数提供了两个不同的内部方法： `[[Call]]` 与 `[[Construct]]` 。当未使用 `new` 进行函数调用时， `[[Call]]` 方法会被执行，运行的是代码中的函数体。而当使用 `new` 进行函数调用时， `[[Construct]]` 方法则会被执行，负责创建一个被称为新目标的新对象，并且将该新目标作为 `this` 去执行函数体。拥有 `[[Construct]]` 方法的函数被称为构造器。

> 切记并非所有函数都拥有 `[[Construct]]` 方法，因此不是所有函数都可以用 `new` 去调用。在后面会介绍的箭头函数就是个例外。

在 ES5 中判断函数如何被调用

在 ES5 中判断是否使用了 `new` 去调用函数（即作为构造器），最流行的方式是使用 `instanceof` ，例如：

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;   // 使用 new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");  // 抛出错误
```

此处对 `this` 值进行了检查，来判断其是否为构造器的一个实例：若是，正常继续执行；否则抛出错误。这么做能奏效是因为 `[[Construct]]` 方法创建了 `Person` 的一个新实例并将其赋值给 `this` 。可惜该方法并不绝对可靠，因为有时未使用 `new` 但 `this` 仍然可能是 `Person` 的实例，正如下例：

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;   // 使用 new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // 奏效了！
```

调用 `Person.call()` 并将 `person` 变量作为第一个参数传入，这意味着将 `Person` 内部的 `this` 被设置为了 `person` 。对于该函数来说，没任何办法能将这种方式与使用 `new` 调用区分开来。

new.target 元属性

为了解决这个问题，ES6 引入了 `new.target` 元属性。元属性指的是“非对象”（例如 `new` 运算符）上的属性，并提供关联目标的附加信息。当函数的 `[[Construct]]` 方法被调用时，`new` 运算符的作用目标会填入 `new.target` 元属性，此时函数体内部的 `this` 值是新创建的对象实例，而 `new.target` 的值正是该实例的构造器。而若 `[[Call]]` 被执行，`new.target` 的值将会是 `undefined` 。

通过检查 `new.target` 这个新的元属性是否被定义，就能让你安全地判断函数被调用时是否使用了 `new` 。

```
function Person(name) {
    if (typeof new.target !== "undefined") {
        this.name = name;   // 使用 new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // 出错！
```

使用 `new.target` 而非 `this instanceof Person` ，`Person` 构造器会在未使用 `new` 时正确地抛出错误。

也可以检查 `new.target` 来判断是否使用特定构造器进行了调用，例如以下代码：

```
function Person(name) {
    if (new.target === Person) {
        this.name = name;   // 使用 new
    } else {
        throw new Error("You must use new with Person.")
    }
}

function AnotherPerson(name) {
    Person.call(this, name);
}

var person = new Person("Nicholas");
var anotherPerson = new AnotherPerson("Nicholas");  // 出错！
```

在此代码中，为了正确工作，`new.target` 必须是 `Person` 。当调用 `new AnotherPerson("Nicholas")` 时，`Person.call(this, name)` 也随之被调用，从而抛出了错误，因为此时在 `Person` 构造器内部的 `new.target` 值为 `undefined` （调用 `Person` 时并未使用 `new` ）。

> 警告：在函数之外使用 `new.target` 会导致语法错误。

ES6 通过新增 `new.target` 而消除了函数调用方式的不确定性。在此方面 ES6 还随之解决了语言此前另一个不确定的部分——在代码块内部声明函数。

块级函数

在 ES3 或更早版本中，在代码块中声明函数（即块级函数）严格来说应当是一个语法错误，但所有的浏览器却都支持该语法。可惜每个浏览器都有轻微的行为差异，所以最佳实践就是切勿在代码块中声明函数，更好的选择是使用函数表达式。

为了控制这种不兼容行为， ES5 的严格模式为代码块内部的函数声明引入了一种错误，就像这样：

```
"use strict";

if (true) {

    // 在 ES5 会抛出语法错误， ES6 则不会
    function doSomething() {
        // ...
    }
}
```

在 ES5 中，这段代码会抛出语法错误。不过 ES6 会将 `doSomething()` 函数视为块级声明，并允许它在声明所在的代码块内部被访问。例如：

```
"use strict";

if (true) {

    console.log(typeof doSomething);        // "function"

    function doSomething() {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);            // "undefined"
```

块级函数会被提升到所在代码块的顶部，因此 `typeof doSomething` 会返回 `"function"` ，即便该检查位于此函数定义位置之前。一旦 `if` 代码块执行完毕， `doSomething()` 也就不复存在。

决定何时使用块级函数

块级函数与 `let` 函数表达式相似，在执行流跳出定义所在的代码块之后，函数定义就会被移除。关键区别在于：块级函数会被提升到所在代码块的顶部；而使用 `let` 的函数表达式则不会，正如以下范例所示：

```
"use strict";

if (true) {

    console.log(typeof doSomething);        // 抛出错误

    let doSomething = function () {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);
```

此处代码在 `typeof doSomething` 被执行时中断了，因为 `let` 声明尚未被执行，`doSomething()` 位于暂时性死区。了解这个区别之后，就能根据是否想要提升，来选择应当使用块级函数还是 `let` 表达式。

非严格模式的块级函数

ES6 在非严格模式下同样允许使用块级函数，但行为有细微不同。块级函数的作用域会被提升到所在函数或全局环境的顶部，而不是代码块的顶部。

```
// ES6 behavior
if (true) {

    console.log(typeof doSomething);        // "function"

    function doSomething() {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);            // "function"
```

本例中的 `doSomething()` 会被提升到全局作用域，因此在 `if` 代码块外部它仍然存在。ES6 标准化了这种行为，以便移除浏览器此前存在的不兼容性，于是在所有 ES6 运行环境中其行为都会遵循相同的方式。

允许使用块级函数只是增强了在 JS 中声明函数的能力，但 ES6 还引入了一种全新的函数声明方式。

箭头函数

ES6 最有趣的一个新成分便是箭头函数（ **arrow function** ）。箭头函数正如名称所示那样使用一个“箭头”（ `=>` ）来定义，但它的行为在很多重要方面与传统的 JS 函数不同：

- 没有 **`this`** 、 **`super`** 、 **`arguments`** ，也没有 **`new.target`** 绑定： `this` 、 `super` 、 `arguments` 、以及函数内部的 `new.target` 的值由外层最近的非箭头函数来决定（ `super` 详见第四章）。
- 不能使用 **`new`** 去调用：箭头函数没有 `[[Construct]]` 方法，因此不能被用为构造函数，使用 `new` 调用箭头函数会抛出错误。
- 没有原型： 既然不能对箭头函数使用 `new` ，那么它也不需要原型，也就是没有 `prototype` 属性。
- 不能更改 **`this`** ： `this` 的值在函数内部不能被修改，在函数的整个生命周期内其值会保持不变。
- 没有 **`arguments`** 对象：箭头函数没有 `arguments` 绑定，你必须依赖于具名参数或剩余参数来访问函数的参数。
- 不允许重复的具名参数：箭头函数不允许拥有重复的具名参数，无论是否在严格模式下；而相对来说，传统函数只有在严格模式下才禁止这种重复。

产生这些差异是有理由的。首先并且最重要的是，在 JS 编程中 `this` 绑定是发生错误的常见根源之一，在嵌套的函数中有时会因为调用方式的不同，而导致丢失对外层 `this` 值的追踪，就可能会导致预期外的程序行为。其次，箭头函数使用单一的 `this` 值来执行代码，使得 JS 引擎可以更容易对代码的操作进行优化；而常规函数可能会作为构造函数使用，导致 `this` 易变而不利优化。

其余差异也聚集在减少箭头函数内部的错误与不确定性，这样 JS 引擎也能更好地优化箭头函数的运行。

> 注意：箭头函数也拥有 `name` 属性，并且遵循与其他函数相同的规则。

箭头函数语法

箭头函数的语法可以有多种变体，取决于你要完成的目标。所有变体都以函数参数为开头，紧跟着的是箭头，再接下来则是函数体。参数与函数体都根据实际使用有不同的形式。例如，以下箭头函数接收单个参数并直接返回它：

```
var reflect = value => value;

// 基本等价于：

var reflect = function(value) {
    return value;
};
```

当箭头函数只有单个参数时，该参数可以直接书写而不需要额外的语法；接下来是箭头以及箭头右边的表达式，该表达式会被计算并返回结果。即使此处没有明确的 `return` 语句，该箭头函数仍然会将所传入的参数返回出来。

如果需要传入多于一个的参数，就需要将它们放在括号内，就像这样：

```
var sum = (num1, num2) => num1 + num2;

// 基本等价于：

var sum = function(num1, num2) {
    return num1 + num2;
};
```

`sum()` 函数简单地将两个参数相加并返回结果。此箭头函数与上面的 `reflect()` 之间唯一区别在于：此处的参数被封闭在括号内，相互之间使用逗号分隔，就像传统函数那样。

如果函数没有任何参数，那么在声明时就必须使用一对空括号，就像这样：

```
var getName = () => "Nicholas";

// 基本等价于：

var getName = function() {
    return "Nicholas";
};
```

当你想使用更传统的函数体、也就是可能包含多个语句的时候，需要将函数体用一对花括号进行包裹，并明确定义一个返回值，正如下面这个版本的 `sum()` ：

```
var sum = (num1, num2) => {
    return num1 + num2;
};

// 基本等价于：

var sum = function(num1, num2) {
    return num1 + num2;
};
```

除了 `arguments` 对象不可用之外，几乎可将花括号内部的代码当做传统函数那样对待。

若你想创建一个空函数，就必须使用空的花括号，就像这样：

```
var doNothing = () => {};

// 基本等价于：

var doNothing = function() {};
```

花括号被用于表示函数的主体，它在你至今看到的例子中都工作正常。但若要箭头函数向外返回一个对象字面量，就必须将该字面量包裹在圆括号内，例如：

```
var getTempItem = id => ({ id: id, name: "Temp" });

// 基本等价于：

var getTempItem = function(id) {

    return {
        id: id,
        name: "Temp"
    };
};
```

将对象字面量包裹在括号内，标示了括号内是一个字面量而不是函数体。

创建立即调用函数表达式

JS 中使用函数的一种流行方式是创建立即调用函数表达式（ immediately-invoked function expression ， IIFE ）。 IIFE 允许你定义一个匿名函数并在未保存引用的情况下立刻调用它。当你想创建一个作用域并与程序其他部分相隔离时，这种模式就很有用。例如：

```
let person = function(name) {

    return {
        getName: function() {
            return name;
        }
    };

}("Nicholas");

console.log(person.getName());      // "Nicholas"
```

此代码中 IIFE 被用于创建一个包含 `getName()` 方法的对象。该方法使用 `name` 参数作为返回值，实际上让 `name` 成为所返回对象的一个私有成员。

你可以使用箭头函数来完成同样的事情，只要将其包裹在括号内即可：

```
let person = ((name) => {

    return {
        getName: function() {
            return name;
        }
    };

})("Nicholas");

console.log(person.getName());      // "Nicholas"
```

需要注意的是括号仅包裹了箭头函数的定义，并未包裹 `("Nicholas")` 。使用传统函数时，括号既可以连函数定义与参数调用一起包裹，也可以只用于包裹函数定义，箭头函数与此有别。

> 译注：使用传统函数时， `(function(){/*函数体*/})();` 与 `(function(){/*函数体*/}());` 这两种方式都是可行的。
>
> 但若使用箭头函数，则只有下面的写法是有效的： `(() => {/*函数体*/})();`

没有 this 绑定

JS 最常见的错误领域之一就是在函数内的 `this` 绑定。由于一个函数内部的 `this` 值会根据调用该函数时的上下文而改变，因此完全可能违背本意地影响了预期外的对象。研究如下例子：

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click", function(event) {
            this.doSomething(event.type);     // 错误
        }, false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

此代码的 `PageHandler` 对象被设计用于处理页面上的交互。调用 `init()` 方法以建立交互，并注册了一个事件处理函数来调用 `this.doSomething()` 。然而此代码并未按预期工作。

此处的 `this` 是对事件目标对象（也就是 `document` ）的一个引用，而没有绑定到 `PageHandler` 上，因此调用 `this.doSomething()` 会被中断。若试图运行此代码，你将会在事件处理函数被触发时得到一个错误，因为 `document` 对象并不存在 `doSomething()` 方法。

你可以明确使用 `bind()` 方法将函数的 `this` 值绑定到 `PageHandler` 上，以修正这段代码，就像这样：

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click", (function(event) {
            this.doSomething(event.type);     // 没有错误
        }).bind(this), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

现在此代码能像预期那样运行，但看起来有点奇怪。通过调用 `bind(this)` ，你实际上创建了一个新函数，它的 `this` 被绑定到当前 `this` ，也就是 `PageHandler` 。为了避免额外创建一个函数，修正此代码的更好方式是使用箭头函数。

箭头函数没有 `this` 绑定，意味着箭头函数内部的 `this` 值只能通过查找作用域链来确定。如果箭头函数被包含在一个非箭头函数内，那么 `this` 值就会与该函数的相等；否则， `this` 值就会是全局对象（在浏览器中是 `window` ，在 nodejs 中是 `global` ）。你可以使用

箭头函数来书写如下代码：

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click",
                event => this.doSomething(event.type), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

本例中的事件处理函数是一个调用 `this.doSomething()` 的箭头函数，它的 `this` 值与 `init()` 方法的相同，因此这个版本的代码的工作方式与使用了 `bind(this)` 的上个例子相似。尽管 `doSomething()` 方法并不返回任何值，它仍然是函数体内唯一被执行的语句，因此无需使用花扩花来包裹它。

箭头函数被设计为“抛弃型”的函数，因此不能被用于定义新的类型； `prototype` 属性的缺失让这个特性显而易见。对箭头函数使用 `new` 运算符会导致错误，正如下例：

```
var MyType = () => {},
    object = new MyType();  // 错误：你不能对箭头函数使用 'new'
```

由于 `MyType()` 是一个箭头函数，它就不存在 `[[Construct]]` 方法，此代码调用 `new MyType()` 的操作也因此失败。了解箭头函数不能被用于 `new` 的特性后， JS 引擎就能进一步对其进行优化。

同样，由于箭头函数的 `this` 值由包含它的函数决定，因此不能使用 `call()` 、 `apply()` 或 `bind()` 方法来改变其 `this` 值。

箭头函数与数组

箭头函数的简洁语法也让它成为进行数组操作的理想选择。例如，若你想使用自定义比较器来对数组进行排序，通常会这么写：

```
var result = values.sort(function(a, b) {
    return a - b;
});
```

这里为一个非常简单的工序使用了过多代码，可以比较一下使用了箭头函数的更简洁版本：

```
var result = values.sort((a, b) => a - b);
```

能使用回调函数的数组方法，例如 `sort()` 、 `map()` 与 `reduce()` 方法，都能从箭头函数的简洁语法中获得收益，它用简单的代码实现看似复杂的需求。

没有 arguments 绑定

尽管箭头函数没有自己的 `arguments` 对象，但仍然能访问包含它的函数的 `arguments` 对象。无论此后箭头函数在何处执行，该对象都是可用的。例如：

```
function createArrowFunctionReturningFirstArg() {
    return () => arguments[0];
}

var arrowFunction = createArrowFunctionReturningFirstArg(5);

console.log(arrowFunction());       // 5
```

在 `createArrowFunctionReturningFirstArg()` 内部， `arguments[0]` 元素被已创建的箭头函数 `arrowFunction` 所引用，该引用包含了传递给 `createArrowFunctionReturningFirstArg()` 函数的首个参数。当箭头函数在此后被执行时，它返回了 5 ，正是传递给 `createArrowFunctionReturningFirstArg()` 的首个参数。尽管箭头函数 `arrowFunction` 已不在创建它的函数的作用域内，但由于 `arguments` 标识符的作用域链解析，当时的 `arguments` 对象依然可被访问。

识别箭头函数

尽管语法不同，但箭头函数依然属于函数，并能被照常识别。研究如下代码：

```
var comparator = (a, b) => a - b;

console.log(typeof comparator);                 // "function"
console.log(comparator instanceof Function);    // true
```

`console.log()` 的输出揭示了 `typeof` 与 `instanceof` 在作用于箭头函数时的行为，与作用在其他函数上完全一致。

就像对待其他函数那样，你仍然可以对箭头函数使用 `call()` 、 `apply()` 与 `bind()` 方法，只是箭头函数的 `this` 绑定并不会受影响。这里有几个例子：

```
var sum = (num1, num2) => num1 + num2;

console.log(sum.call(null, 1, 2));       // 3
console.log(sum.apply(null, [1, 2]));   // 3

var boundSum = sum.bind(null, 1, 2);

console.log(boundSum());                // 3
```

`sum()` 函数被使用 `call()` 与 `apply()` 方法调用并传入了参数。 `bind()` 方法被用于创建 `boundSum()` ，后者的两个参数已被绑定为 `1` 与 `2` ，因此调用时不再需要传入这两个参数。

箭头函数能在任意位置，包括使用回调函数时，替代你当前使用的匿名函数。下一节介绍的内容是 ES6 的另一项主要改进，不过该内容完全是内部实现，并没有使用新语法。

尾调用优化

在 ES6 中对函数最有趣的改动或许就是一项引擎优化，它改变了尾部调用的系统。尾调用（ **tail call** ）指的是调用函数的语句是另一个函数的最后语句，就像这样：

```
function doSomething() {
    return doSomethingElse();   // 尾调用
}
```

在 ES5 引擎中实现的尾调用，其处理与其他函数调用一致：一个新的栈帧（ stack frame ）被创建并推到调用栈之上，用于表示该次函数调用。这意味着之前每个栈帧都被保留在内存中，当调用栈过大时会出问题。

有何不同？

ES6 在严格模式下力图为特定尾调用减少调用栈的大小，非严格模式的尾调用则保持不变。当满足以下条件时，尾调用优化不会创建新的栈帧，而是清除当前栈帧并再次利用它：

1. 尾调用不能引用当前栈帧中的变量（意味着该函数不能是闭包）；
2. 进行尾调用的函数在尾调用返回结果后不能做额外操作；
3. 尾调用的结果作为当前函数的返回值。

作为一个例子，下面代码满足了全部三个条件，因此能被轻易地优化：

```
"use strict";

function doSomething() {
    // 被优化
    return doSomethingElse();
}
```

该函数对 `doSomethingElse()` 进行了一次尾调用，并立即返回了其结果，同时并未访问局部作用域的任何变量。但若做个小小改动，不返回结果，就会产生一个无法被优化的函数：

```
"use strict";

function doSomething() {
    // 未被优化：缺少 return
    doSomethingElse();
}
```

类似的，如果你的函数在尾调用返回结果之后进行了额外操作，那么该函数也无法被优化：

```
"use strict";

function doSomething() {
    // 未被优化：在返回之后还要执行加法
    return 1 + doSomethingElse();
}
```

此例在 `doSomethingElse()` 的结果上对其进行了加 1 操作，而没有直接返回该结果，这足以关闭优化。

无意中关闭优化的另一个常见方式，是将函数调用的结果储存在一个变量上，之后才返回了结果，就像这样：

```
"use strict";

function doSomething() {
    // 未被优化：调用并不在尾部
    var result = doSomethingElse();
    return result;
}
```

本例之所以不能被优化，是因为 `doSomethingElse()` 的值并没有立即被返回。

使用闭包或许就是需要避免的最困难情况，因为闭包能够访问外层作用域的变量，会导致尾调用优化被关闭。例如：

```
"use strict";

function doSomething() {
    var num = 1,
        func = () => num;

    // 未被优化：此函数是闭包
    return func();
}
```

此例中闭包 `func()` 需要访问局部变量 `num` ，虽然调用 `func()` 后立即返回了其结果，但是对于 `num` 的引用导致优化不会发生。

如何控制尾调用优化

在实践中，尾调用优化由引擎进行，除非要尽力去优化一个函数，否则不必对此考虑太多。尾调用优化的主要用例是在递归函数中，而且此时的优化能达到最大效果。考虑以下计算阶乘的函数：

```
function factorial(n) {

    if (n <= 1) {
        return 1;
    } else {

        // 未被优化：在返回之后还要执行乘法
        return n * factorial(n - 1);
    }
}
```

此版本的函数并不会被优化，因为在递归调用 `factorial()` 之后还要执行乘法运算。如果 `n` 是一个大数字，那么调用栈的大小会持续增长，并且可能导致堆栈溢出。

为了优化此函数，你需要确保在最后的函数调用之后不会发生乘法运算。为此可以使用一个默认参数来将乘法操作移出 `return` 语句。有结果的函数携带着临时结果进入下一次迭代，这样创建的函数的功能与前例相同，但它能被 ES6 的引擎所优化。此处是新的代码：

```
function factorial(n, p = 1) {

    if (n <= 1) {
        return 1 * p;
    } else {
        let result = n * p;

        // 被优化
        return factorial(n - 1, result);
    }
}
```

在重写的 `factorial()` 函数中，添加了默认值为 1 的第二个参数 `p` 。 `p` 参数保存着前一次乘法的结果，因此下一次的结果就能在进行函数调用之前被算出。当 `n` 大于 1 时，会先进行乘法运算并将其结果作为第二个参数传入 `factorial()` 。这就允许 ES6 引擎去优化这个递归调用。

尾调用优化是你在书写任意递归函数时都需要考虑的因素，因为它能提供显著的性能提升，尤其是被应用到计算复杂度很高的函数时。

总结

函数在 ES6 中并未经历巨大变化，然而一系列增量改进使得函数更易使用。

在特定参数未被传入时，函数的默认参数允许你更容易指定需要使用的值。而在 ES6 之前，为了检查参数是已否已提供，并在未提供时为其分配一个预设值，需要在函数内书写一些额外代码。

剩余参数允许你将余下的所有参数放入指定数组。使用真正的数组，并可以根据需要指定包含哪些参数，让剩余参数成为比 `arguments` 更为灵活的解决方案。

扩展运算符是剩余参数的好伙伴，允许在调用函数时将数组解构为分离的参数。在 ES6 之前，要将数组的元素用作独立参数传给函数只有两种办法：手动指定每一个参数，或者使用 `apply()` 方法。有了扩展运算符，你就能轻易将数组传递给函数而无须顾虑该函数的 `this` 绑定。

新增的 `name` 属性让你在调试与执行方面能更容易地识别函数。此外， ES6 正式定义了块级函数的行为，因此在严格模式下它们不再是语法错误。

在 ES6 中，函数的行为被 `[[Call]]` 与 `[[Construct]]` 方法所定义，前者对应普通的函数执行，后者则对应着使用了 `new` 的调用。 `new.target` 元属性也能用于判断函数被调用时是否使用了 `new` 。

ES6 函数的最大变化就是增加了箭头函数。箭头函数被设计用于替代匿名函数表达式，它拥有更简洁的语法、词法级的 `this` 绑定，并且没有 `arguments` 对象。此外，箭头函数不能修改它们的 `this` 绑定，因此不能被用作构造器。

尾调用优化允许某些函数的调用被优化，以保持更小的调用栈、使用更少的内存，并防止堆栈溢出。当能进行安全优化时，它会由引擎自动应用。不过你可以考虑重写递归函数，以便利用这种优化。

第四章 扩展的对象功能

ES6 注重于提高对象的效用，这是因为在 JS 中几乎所有的值都是某种类型的对象。此外，随着 JS 应用复杂度的增长，在 JS 程序中所使用的对象的平均数也在持续增长，因此更有效使用对象就变得更加必要。

ES6 在对象的许多方面都进行了改进，从简单的语法扩展，到操作与交互。

对象类别

相对于那些由浏览器或 Node.js 之类运行环境所添加的对象， JS 使用混合术语来描述在标准中的对象， ES6 规范明确定义了对象的每种类别。理解对象术语对于从整体上清楚认识这门语言来说非常重要。对象类别包括：

- 普通对象：拥有 JS 对象所有默认的内部行为。
- 奇异对象：其内部行为在某些方面有别于默认行为。
- 标准对象：在 ES6 中被定义的对象，例如 `Array` 、 `Date` ，等等。标准对象可以是普通的，也可以是奇异的。
- 内置对象：脚本在 JS 运行环境中开始运行时可用的对象。所有的标准对象都是内置对象。

我会在整本书中使用这些术语来讲解在 ES6 中定义的各种对象。

对象字面量语法的扩展

对象字面量是 JS 中最流行的模式之一，它存在于互联网上的几乎所有 JS 文件中， JSON 就是基于这种语法。对象字面量如此流行，是因为它能用简洁语法创建对象，避免了书写冗余代码。对开发者来说幸运的是， ES6 用几种方式扩展了对象字面量，让这种语法变得更强大也更简洁。

属性初始化器的速记法

在 ES5 及更早版本中，对象字面量是简单的“键/值对”集合。这意味着初始化属性值的代码可能会有些重复，例如：

```
function createPerson(name, age) {
    return {
        name: name,
        age: age
    };
}
```

`createPerson()` 函数创建了一个对象，其属性名与函数的参数名相同。这看起来重复了 `name` 与 `age` ，尽管左侧是对象属性的名称，而右侧则负责给属性提供值。返回对象的 `name` 键与 `age` 键分别被变量 `name` 与 `age` 变量所赋值。

在 ES6 中，你可以使用属性初始化器的速记法来消除对象名称与本地变量名的重复情况。当对象的一个属性名称与本地变量名相同时，可以简单书写名称而省略冒号与值。例如， `createPerson()` 能用 ES6 这样重写：

```
function createPerson(name, age) {
    return {
        name,
        age
    };
}
```

当对象字面量中的属性只有名称时， JS 引擎会在周边作用域查找同名变量。若找到，该变量的值将会被赋给对象字面量的同名属性。在本例中，局部变量 `name` 的值就被赋给了 `name` 属性。

这个扩展让对象字面量的初始化更加简洁，也有助于消除命名错误。用局部变量为对象同名属性赋值在 JS 中是极其常见的模式，因此这个扩展自然非常受欢迎。

方法简写

ES6 同样改进了为对象字面量方法赋值的语法。在 ES5 及更早版本中，你必须指定一个名称并用完整的函数定义来为对象添加方法，如下：

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};
```

通过省略冒号与 `function` 关键字，ES6 将这个语法变得更简洁，这意味着你可以这样重写上个例子：

```
var person = {
    name: "Nicholas",
    sayName() {
        console.log(this.name);
    }
};
```

这种速记语法也被称为方法简写语法（ **concise method** syntax ），与上例一样在 `person` 对象中创建了一个方法。 `sayName()` 属性被一个匿名函数所赋值，并且具备 ES5 版本的 `sayName()` 方法的所有特征。但有一点区别是：方法简写能使用 `super` ，而非简写的方法则不能（ `super` 会在后面的小节中讨论）。

> 使用方法简写速记法创建的方法，其 `name` 属性就是括号之前的名称。上面这个例子中， `person.sayName()` 的名称属性就是 `"sayName"` 。

可计算属性名

在 ES5 及更早版本中，对象实例能使用“可计算的属性名”，只要用方括号表示法来代替小数点表示法即可。方括号允许你指定变量或字符串字面量为属性名，并且在字符串中允许存在不能用于标识符的特殊字符。此处有个范例：

```
var person = {},
    lastName = "last name";

person["first name"] = "Nicholas";
person[lastName] = "Zakas";

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

`lastName` 变量的值为 `"last name"` ，这样该例中两个属性名都包含了空格，这种情况是无法使用小数点表示法的，而方括号表示法允许将任意字符串用作属性名。最终 `"first name"` 与 `"last name"` 属性分别被赋值为 `"Nicholas"` 与 `"Zakas"` 。

此外，你可以在对象字面量中直接使用字符串字面量作为属性，就像这样：

```
var person = {
    "first name": "Nicholas"
};

console.log(person["first name"]);      // "Nicholas"
```

这种模式要求属性名事先已知、并且能用字符串字面量表示。然而，若属性名被包含在变量中（就像前面例子中的 `"first name"` ），或者必须通过计算才能获得，那么在 ES5 中就无法为对象字面量定义这种属性。

在 ES6 中，可计算属性名是对象字面量语法的一部分，它用的也是方括号表示法，与此前在对象实例上的用法一致。例如：

```
var lastName = "last name";

var person = {
    "first name": "Nicholas",
    [lastName]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

对象字面量内的方括号表明该属性名需要计算，其结果是一个字符串。这意味着其中可以包含表达式，像下面这样：

```
var suffix = " name";

var person = {
    ["first" + suffix]: "Nicholas",
    ["last" + suffix]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person["last name"]);       // "Zakas"
```

这些属性名被计算为 `"first name"` 与 `"last name"` ，而这两个字符串此后可以用来引用对应属性。使用了方括号表示法，任何能放在对象实例方括号内的东西，都可以作为可计算属性名用在对象字面量中。

新的方法

从 ES5 开始就有这么一个设计意图：避免创建新的全局函数，避免在 `Object` 对象的原型上添加新方法，而尽量尝试将新方法添加到合适对象上。不过，当新方法不适用于其他任何对象时，全局的 `Object` 对象就会收到越来越多的方法。 ES6 也在 `Object` 对象上引入了两个新方法，以便让特定任务更易完成。

Object.is() 方法

在 JS 中当要比较两个值时，你可能会使用相等运算符（ `==` ）或严格相等运算符（ `===` ）。为了避免在比较时发生强制类型转换，许多开发者更倾向于使用后者。但严格相等运算符也并不完全准确，例如，它认为 `+0` 与 `-0` 相等，尽管这两者在 JS 引擎中有不同的表示；另外 `NaN === NaN` 会返回 `false` ，因此只有用 `isNaN()` 函数才能正确检测 `NaN` 。

ES6 引入了 `Object.is()` 方法来弥补严格相等运算符残留的怪异缺陷。此方法接受两个参数，并会在二者类型相同并且值也相等相等时返回 `true` 。此处有个例子：

```
console.log(+0 == -0);              // true
console.log(+0 === -0);             // true
console.log(Object.is(+0, -0));     // false

console.log(NaN == NaN);            // false
console.log(NaN === NaN);           // false
console.log(Object.is(NaN, NaN));   // true

console.log(5 == 5);                // true
console.log(5 == "5");              // true
console.log(5 === 5);               // true
console.log(5 === "5");             // false
console.log(Object.is(5, 5));       // true
console.log(Object.is(5, "5"));     // false
```

绝大多数情况下， `Object.is()` 的结果与 `===` 运算符是相同的，仅有的例外是：它会认为 `+0` 与 `-0` 不相等，而且 `NaN` 等于 `NaN` 。但没必要停止使用严格相等运算符，选择 `Object.is()` 还是选择 `==` 或 `===` ，取决于代码的实际需求。

Object.assign() 方法

混入（ **Mixin** ）是在 JS 中组合对象时最流行的模式。在一次混入中，一个对象会从另一个对象中接收属性与方法。很多 JS 的库中都有类似下面这样的混入方法：

```
function mixin(receiver, supplier) {
    Object.keys(supplier).forEach(function(key) {
        receiver[key] = supplier[key];
    });

    return receiver;
}
```

`mixin()` 函数在 `supplier` 对象的自有属性上进行迭代，并将这些属性复制到 `receiver` 对象（浅复制，当属性值为对象时，仅复制其引用）。这样 `receiver` 对象就能获得新的属性而无须使用继承，正如下面代码：

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
};

var myObject = {};
mixin(myObject, EventTarget.prototype);

myObject.emit("somethingChanged");
```

此处 `myObject` 对象接收了 `EventTarget.prototype` 对象的行为，获得了使用 `emit()` 方法来发布事件、使用 `on()` 来订阅事件的能力。

此模式非常流行，于是 ES6 也添加了 `Object.assign()` 方法来完成同样的行为。该方法接受一个接收者对象，以及任意数量的源对象，并会返回接收者对象。由于像前面这样的 `mixin()` 函数使用了赋值运算符（ `=` ），也就无法将访问器属性复制到接收者上，而 `assign()` 更能反映出实际发生的操作，因此 `mixin()` 这个名称被弃用了。

> 各式各样的库中都有相似但名称不同的方法，流行的有 `extend()` 或 `mix()` ，其基本功能都相同。而 ES6 草案除了 `Object.assign()` 之外，也曾短暂存在一个 `Object.mixin()` 方法，二者的主要差异在于 `Object.mixin()` 也会复制访问器属性。但考虑到 `super` 的使用，此方法最终被移除了（本章后面小节会介绍 `super` ）。

你可以在任意曾使用 `mixin()` 函数的地方使用 `Object.assign()` ，此处有个例子：

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
}

var myObject = {}
Object.assign(myObject, EventTarget.prototype);

myObject.emit("somethingChanged");
```

`Object.assign()` 方法接受任意数量的源对象，而接收对象会按照源对象在参数中的顺序来依次接收它们的属性。这意味着在接收对象中，后面源对象的属性可能会覆盖前面的，下面代码片段中就有这种情况：

```
var receiver = {};

Object.assign(receiver,
    {
        type: "js",
        name: "file.js"
    },
    {
        type: "css"
    }
);

console.log(receiver.type);     // "css"
console.log(receiver.name);     // "file.js"
```

`receiver.type` 的值为 `"css"` ，这是因为第二个源对象覆盖了第一个源对象的值。

`Object.assign()` 方法并不是 ES6 的一项重大扩展，但它确实把很多 JS 库中存在的一个公共方法标准化了。

> 操作访问器属性
>
> 切记 `Object.assign()` 不能将源对象的访问器属性复制到接收对象中，由于它使用了赋值运算符，源对象的访问器属性就会转变成接收对象的数据属性，例如：
>
> ```
> var receiver = {},
> supplier = {
>   get name() {
>       return "file.js"
>   }
> };
>
> Object.assign(receiver, supplier);
>
> var descriptor = Object.getOwnPropertyDescriptor(receiver, "name");
>
> console.log(descriptor.value);      // "file.js"
> console.log(descriptor.get);        // undefined
> ```
>
> 此代码中的 `supplier` 对象拥有一个名为 `name` 的访问器属性。在使用 `Object.assign()` 方法时， `supplier.name` 返回的值是 `"file.js"` ，于是该值就被存储到 `receiver.name` 数据属性上。

重复的对象字面量属性

ES5 严格模式为重复的对象字面量属性引入了一个检查，存在重复的属性名时会抛出错误。例如，以下代码就有问题：

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"        // 在 ES5 严格模式中是语法错误
};
```

在 ES5 严格模式下运行时，第二个 `name` 属性会造成语法错误。但 ES6 移除了重复属性的检查，无论是否严格模式，都不再进行检查。当存在重复属性时，排在后面的属性的值会成为该属性的实际值，如下所示：

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"        // 在 ES6 严格模式中不会出错
};

console.log(person.name);       // "Greg"
```

在本例中， `person.name` 的值为 `"Greg"` ，因为这是赋给该属性的最后一个值。

自有属性的枚举顺序

ES5 并没有定义对象属性的枚举顺序，而是将该问题留给了 JS 引擎厂商。 ES6 则严格定义了对象自有属性在被枚举时的返回顺序。这影响了 `Object.getOwnPropertyNames()` 与 `Reflect.ownKeys` （详见第十二章）返回属性的方式，也同样影响了 `Object.assign()` 处理属性的顺序。

自有属性枚举时基本顺序如下：

1. 所有的数字类型键，按升序排列。
2. 所有的字符串类型键，按被添加到对象的顺序排列。
3. 所有的符号类型（详见第六章）键，也按添加顺序排列。

此处有个示例：

```
var obj = {
    a: 1,
    0: 1,
    c: 1,
    2: 1,
    b: 1,
    1: 1
};

obj.d = 1;

console.log(Object.getOwnPropertyNames(obj).join(""));     // "012acbd"
```

`Object.getOwnPropertyNames()` 方法按 `0` 、 `1` 、 `2` 、 `a` 、 `c` 、 `b` 、 `d` 的顺序返回了 `obj` 对象的属性。注意，数值类型的键不会遵循在对象字面量中的顺序，而会被合并及排序。字符串类型的键会跟在数值类型的键之后，按照被添加到 `obj` 对象的顺序，在对象字面量中定义的键会首先出现，接下来是此后动态添加到对象的键。

> 对于 `for-in` 循环，并非所有的 JS 引擎都采用相同的处理方式，其枚举顺序仍未被明确规定。而 `Object.keys()` 和 `JSON.stringify()` 也使用了与 `for-in` 一样的枚举顺序。

虽然枚举顺序的变动对 JS 的工作方式影响甚小，但依赖于特定枚举顺序才能正确运行的程序并不罕见。 ES6 通过规定枚举的顺序，就确保了依赖枚举操作的 JS 代码无论在什么运行环境中都能正常工作。

更强大的原型

原型是在 JS 中进行继承的基础， ES6 则继续强化原型。早期的 JS 版本对原型的使用有严重限制，然而随着语言的成熟，开发者也越来越熟悉原型的工作机制，因此他们明确希望能对原型有更多控制权、并能更方便地使用它。于是 ES6 就给原型引入了一些改进。

修改对象的原型

一般来说，对象的原型会在通过构造器或 `Object.create()` 方法创建对象时被指定。 JS 编程到 ES5 为止最重要的假定之一就是：对象的原型在初始化完成后会保持不变。 ES5 添加了 `Object.getPrototypeOf()` 方法来获取任意指定对象的原型，不过仍然缺少在初始化之后更改对象原型的标准方法。

ES6 通过添加 `Object.setPrototypeOf()` 方法而改变了这种假定。此方法允许你修改任意指定对象的原型，它接受两个参数：需要被修改原型的对象，以及将会成为前者原型的对象。例如：

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

let dog = {
    getGreeting() {
        return "Woof";
    }
};

// 原型为 person
let friend = Object.create(person);
console.log(friend.getGreeting());                      // "Hello"
console.log(Object.getPrototypeOf(friend) === person);  // true

// 将原型设置为 dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());                      // "Woof"
console.log(Object.getPrototypeOf(friend) === dog);     // true
```

此代码定义了两个基础对象： `person` 与 `dog` ，二者都拥有一个名为 `getGreeting()` 的方法，用于返回一个字符串。 `friend` 对象起初继承了 `person` 对象，意味着 `friend.getGreeting()` 方法会输出 `"Hello"` ；当它的原型被更改为 `dog` 对象，与 `person` 的关联就被破坏， `friend.getGreeting()` 方法会改而输出 `"Woof"` 。

对象原型的实际值被存储在一个内部属性 `[[Prototype]]` 上， `Object.getPrototypeOf()` 方法会返回此属性存储的值，而 `Object.setPrototypeOf()` 方法则能够修改该值。不过，使用 `[[Prototype]]` 属性的方式还不止这些。

使用 super 引用的简单原型访问

正如前面所提到的，原型对 JS 来说非常重要，而 ES6 也进行了很多工作来让它更易用。关于原型的另一项进步就是引入了 `super` 引用，这样就能更轻易地在对象原型上进行功能调用。例如，若要覆盖对象实例的一个方法，但依然想调用原型上的同名方法，你可能会这么做：

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

let dog = {
    getGreeting() {
        return "Woof";
    }
};

let friend = {
    getGreeting() {
        return Object.getPrototypeOf(this).getGreeting.call(this) + ", hi!";
    }
};

// 将原型设置为 person
Object.setPrototypeOf(friend, person);
console.log(friend.getGreeting());                      // "Hello, hi!"
console.log(Object.getPrototypeOf(friend) === person);  // true

// 将原型设置为 dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());                      // "Woof, hi!"
console.log(Object.getPrototypeOf(friend) === dog);     // true
```

本例中 `friend` 上的 `getGreeting()` 调用了对象上的同名方法。 `Object.getPrototypeOf()` 方法确保了能调用正确的原型，并在其返回结果上附加了一个字符串；之后的 `call(this)` 则能确保正确设置原型方法内部的 `this` 值。

调用原型上的方法时，要牢记使用 `Object.getPrototypeOf()` 与 `.call(this)` ，实在有点复杂，因此 ES6 才引入了 `super` 。简单来说， `super` 是指向当前对象的原型的一个指针，实际上就是 `Object.getPrototypeOf(this)` 的值。了解这些后，你就可以像下面这样简化 `getGreeting()` 方法：

```
let friend = {
    getGreeting() {
        // 这相当于上个例子中的：
        // Object.getPrototypeOf(this).getGreeting.call(this)
        return super.getGreeting() + ", hi!";
    }
};
```

此处调用 `super.getGreeting()` 等同于在上例的环境中使用 `Object.getPrototypeOf(this).getGreeting.call(this)` 。类似的，你能使用 `super` 引用来调用对象原型上的任何方法，只要这个引用是位于简写的方法之内。试图在方法简写之外的情况使用 `super` 会导致语法错误，正如下例：

```
let friend = {
    getGreeting: function() {
        // 语法错误
        return super.getGreeting() + ", hi!";
    }
};
```

此例使用了一个匿名函数作为属性，于是调用 `super.getGreeting()` 就导致了语法错误，因为在这种上下文中 `super` 是不可用的。

当使用多级继承时， `Object.getPrototypeOf()` 不再适用于所有场景，此时 `super` 引用就能体现出它的强大。例如：

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// 原型为 person
let friend = {
    getGreeting() {
        return Object.getPrototypeOf(this).getGreeting.call(this) + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

// 原型为 friend
let relative = Object.create(friend);

console.log(person.getGreeting());                  // "Hello"
console.log(friend.getGreeting());                  // "Hello, hi!"
console.log(relative.getGreeting());                // error!
```

使用 `Object.getPrototypeOf()` ，在调用 `relative.getGreeting()` 时发生了错误。这是因为此时 `this` 的值是 `relative` ，而 `relative` 的原型是 `friend` 对象，这样 `friend.getGreeting().call()` 调用就会导致进程开始反复进行递归调用，直到发生堆栈错误。

此问题在 ES5 中很难解决，但若使用 ES6 的 `super` ，就很简单了：

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// 原型为 person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

// 原型为 friend
let relative = Object.create(friend);

console.log(person.getGreeting());                  // "Hello"
console.log(friend.getGreeting());                  // "Hello, hi!"
console.log(relative.getGreeting());                // "Hello, hi!"
```

由于 `super` 引用并非是动态的，它总是能指向正确的对象。在本例中，`super.getGreeting()` 总是指向 `person.getGreeting()` ，而不管有多少对象继承了此方法。

正式的“方法”定义

在 ES6 之前，“方法”的概念从未被正式定义，它此前仅指对象的函数属性而非数据属性。ES6 则正式将方法定义为：一个拥有 `[[HomeObject]]` 内部属性的函数，此内部属性指向该方法所属的对象。研究以下例子：

```
let person = {

    // 方法
    getGreeting() {
        return "Hello";
    }
};

// 并非方法
function shareGreeting() {
    return "Hi!";
}
```

此例定义了拥有单个 `getGreeting()` 方法的 `person` 对象。由于 `getGreeting()` 被直接赋给了一个对象，它的 `[[HomeObject]]` 属性值就是 `person` 。而另一方面，`shareGreeting()` 函数被创建时并没有赋给一个对象，它就不具备 `[[HomeObject]]` 属性。这种差异在大多数情况下并不重要，然而使用 `super` 引用时就完全不同了。

任何对 `super` 的引用都会使用 `[[HomeObject]]` 属性来判断要做什么。第一步是在 `[[HomeObject]]` 上调用 `Object.getPrototypeOf()` 来获取对原型的引用；接下来，在该原型上查找同名函数；最后，创建 `this` 绑定并调用该方法。此处有个例子：

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// 原型为 person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

console.log(friend.getGreeting());  // "Hello, hi!"
```

调用 `friend.getGreeting()` 返回了一个字符串，即 `person.getGreeting()` 的返回值与 `", hi!"` 的拼接结果。此时 `friend.getGreeting()` 的 `[[HomeObject]]` 值是 `friend` ，并且 `friend` 的原型是 `person` ，因此 `super.getGreeting()` 就等价于 `person.getGreeting.call(this)` 。

总结

对象是 JS 编程的核心， ES6 对它进行了一些正面改进，让它更易用也更强大。

ES6 为对象字面量做了几个改进。若要将作用域内的变量赋值给对象的同名属性，使用属性定义速记法能简化代码；可计算属性名允许你将非字面量的值指定为属性的名称，类似于此前在其他场合的用法；方法简写让你在对象字面量中定义方法时能省略冒号和 `function` 关键字，从而减少输入的字符数； ES6 还舍弃了对象字面量中重复属性名的检查，意味着你可以在一个对象字面量中书写两个同名属性，而不会抛出错误。

`Object.assign()` 方法让一次性更改单个对象的多个属性变得更加容易，这在使用混入模式时非常有用。 `Object.is()` 方法对任何值都会执行严格相等比较，在处理特殊的 JS 值时，它成为了 `===` 的一个更有效更安全的替代品。

对象自有属性的枚举顺序在 ES6 中被明确定义了。在枚举属性时，数字类型的键总是会首先出现，并按升序排列，此后是字符串类型的键，最后是符号类型的键，后两者都分别按添加顺序排列。

借助 ES6 的 `Object.setPrototypeOf()` 方法，能在对象创建之后再更改其原型。

最后，你能用 `super` 关键字去调用对象原型上的方法，所调用的方法会被设置好其内部的 `this` 绑定，以自动使用该 `this` 值来进行工作。

第五章 解构：更方便的数据访问

对象与数组的字面量在 JS 中是最常用的两种表示法，它们得益于 JSON 数据格式的流行，在语言中的地位越发重要。我们时常要定义对象与数组，然后有条不紊地从这些结构中提取出相关信息。为了简化提取信息的任务， ES6 新增了解构（ **destructuring** ），这是将一个数据结构分解为更小的部分的过程。本章介绍如何在对象与数组上利用解构。

解构为何有用？

在 ES5 及更早版本中，从对象或数组中获取信息，并将特定数据存入本地变量，需要书写许多相似的代码。例如：

```
let options = {
        repeat: true,
        save: false
    };

// 从对象中提取数据
let repeat = options.repeat,
    save = options.save;
```

此代码提取了 `options` 对象的 `repeat` 与 `save` 值，并将其存在同名的本地变量上。这段代码虽然看似简单，但想象一下若有大量变量需要处理，就必须逐个为其赋值；并且若需要遍历一个嵌套的数据结构以寻找信息，有可能会为了一点数据而挖掘整个结构。

这就是 ES6 为何要给对象与数组添加解构。当把数据结构分解为更小的部分时，从中提取所需数据就会变得容易许多。很多语言都能用精简的语法来实现解构，让它更易使用。实际上 ES6 的解构使用了你早已熟悉的语法，也就是对象与数组的字面量语法。

对象解构

对象解构语法在赋值语句的左侧使用了对象字面量，例如：

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

在此代码中， `node.type` 的值被存储到 `type` 本地变量中， `node.name` 的值则存储到 `name` 变量中。此语法与第四章介绍过的属性简写相同。 `type` 与 `name` 标识符既声明了本地变量，也读取了对象的相应属性值。

> 切勿遗忘初始化器
>
> 当使用解构配合 `var` 、 `let` 或 `const` 来声明变量时，必须提供初始化器（即等号右边的值）。下面的代码都会因为缺失初始化器而抛出错误：
>
> ```
> // 语法错误！
> var { type, name };
>
> // 语法错误！
> let { type, name };
>
> // 语法错误！
> const { type, name };
> ```
>
> 无论是否使用解构， `const` 始终要求为变量提供初始化器；而 `var` 与 `let` 则仅在使用解构时才作此要求。

解构赋值

以上对象解构示例都用于变量声明。不过，也可以在赋值的时候使用解构。有时或许会在变量声明之后再更改它们的值，例如：

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;

// 使用解构来分配不同的值
({ type, name } = node);

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

在本例中， `node` 对象的 `type` 与 `name` 属性在声明时被初始化，而两个同名变量也被声明并初始化为其他值。接下来一行使用了解构表达式，通过读取 `node` 对象来更改这两个变量的值。注意你必须用圆括号包裹解构赋值语句，这是因为暴露的花括号会被解析为代码块语句，而块语句不允许在赋值操作符（即等号）左侧出现。圆括号标示了内部的花括号并不是块语句、而应该被解释为表达式，从而完成赋值操作。

解构赋值表达式的值为表达式右侧（在 `=` 之后）的值。也就是说在任何需要使用值的位置，都可以使用解构赋值表达式。例如，传递值给函数：

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;

function outputInfo(value) {
    console.log(value === node);        // true
}

outputInfo({ type, name } = node);

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

`outputInfo()` 函数被调用时使用了一个解构赋值表达式作为参数。该表达式计算结果为 `node` ，也就是表达式右侧的值。对 `type` 与 `name` 的赋值正常进行，同时 `node` 也被传入了 `outputInfo()` 函数。

> 当解构赋值表达式的右侧（ `=` 后面的表达式）的计算结果为 `null` 或 `undefined` 时，会抛出错误。因为任何读取 `null` 或 `undefined` 的属性的企图都会导致“运行时”错误（runtime error）。

默认值

当你使用解构赋值语句时，如果所指定的本地变量在对象中没有找到同名属性，那么该变量会被赋值为 `undefined` 。例如：

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name, value } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // undefined
```

此代码定义了一个额外的本地变量 `value` ，并试图对其赋值。然而 `node` 对象中不存在同名属性，因此 `value` 不出预料地被赋值为 `undefined` 。

你可以选择性地定义一个默认值，以便在指定属性不存在时使用该值。若要这么做，需要在属性名后面添加一个等号并指定默认值，就像这样：

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name, value = true } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // true
```

在此例中，变量 `value` 被指定了一个默认值 `true` ，只有在 `node` 的对应属性缺失、或对应的属性值为 `undefined` 的情况下，该默认值才会被使用。由于此处不存在 `node.value` 属性，变量 `value` 就使用了该默认值。这种工作方式很像函数参数的默认值（详见第三章）。

赋值给非同名本地变量

至此的每个解构赋值示例都使用了对象中的属性名作为本地变量的名称，例如，把 `node.type` 的值存储到 `type` 变量上。使用相同名称时这么做没问题，但若希望使用其他名称呢？ ES6 有一个扩展语法，允许你在给本地变量赋值时使用一个不同的名称，而且该语法看上去就像是使用对象字面量的非简写的属性初始化。此处有个示例：

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type: localType, name: localName } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "foo"
```

此代码使用了解构赋值来声明 `localType` 与 `localName` 变量，分别赋予了 `node.type` 与 `node.name` 属性的值。 `type: localType` 这种语法表示欲读取名为 `type` 的属性，并将它的值存储在变量 `localType` 上。该语法实际上与传统对象字面量语法相反，传统语法将名称放在冒号左侧、值放在冒号右侧；而在本例中，则是名称在右侧，需要读取的位置（对象属性名）则被放在了左侧。

你也可以给变量别名添加默认值，依然是在本地变量名称后添加等号与默认值，例如：

```
let node = {
        type: "Identifier"
    };

let { type: localType, name: localName = "bar" } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "bar"
```

此处的 `localName` 变量拥有一个默认值，由于 `node.name` 属性并不存在，该变量最终被赋为默认值 `"bar"` 。

到此为止，你已经看到如何对属性值为基本类型值的对象进行解构，而对象解构也可被用于从嵌套的对象结构中提取属性值。（嵌套对象表示对象的属性可能还是一个对象）

嵌套的对象解构

使用类似于对象字面量的语法，可以深入到嵌套的对象结构中去提取所需数据。此处有个示例：

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        }
    };

let { loc: { start }} = node;

console.log(start.line);        // 1
console.log(start.column);      // 1
```

本例中的解构模式使用了花括号，表示应当下探到 `node` 对象的 `loc` 属性内部去寻找 `start` 属性。记住上一节介绍过的，每当有一个冒号在解构模式中出现，就意味着冒号之前的标识符代表需要检查的位置，而冒号右侧则是赋值的目标。当冒号右侧存在花括号时，表示目标被嵌套在对象的更深层次中。

更进一步，在对象的嵌套解构中同样能为本地变量使用不同的名称：

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        }
    };

// 提取 node.loc.start
let { loc: { start: localStart }} = node;

console.log(localStart.line);   // 1
console.log(localStart.column); // 1
```

在此版本的代码中， `node.loc.start` 的值被存储到一个新的本地变量 `localStart` 上，解构模式可以被嵌套在任意深度的层级，并且在每个层级的功能都一致。

对象解构十分强大并有很多可用形式，而数组解构则提供了一些独特的能力，用于提取数组中的信息。

> 语法陷阱
>
> 使用嵌套的解构时需要小心，因为你可能无意中就创建了一个没有任何效果的语句。空白花括号在对象解构中是合法的，然而它不会产生任何效果。例如：
>
> ```
> // 没有变量被声明！
> let { loc: {} } = node;
> ```
>
> 在此语句中并未声明任何变量绑定。由于花括号在右侧， `loc` 被作为需检查的位置来使用，并不会创建变量绑定。这种情况更应当用等号来定义一个默认值，而不是用冒号来定义一个位置。这种语法将来可能会被定义为无效语法，然而现在它只是需要规避的一个陷阱。

数组解构

数组解构的语法看起来与对象解构非常相似，只是将对象字面量替换成了数组字面量。数组解构时，解构作用在数组内部的位置上，而不是作用在对象的属性名上，例如：

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

此处数组解构从 `colors` 数组中取出了 `"red"` 与 `"green"` ，并将它们赋值给 `fristColor` 与 `secondColor` 变量。这些值根据它们在数组中的位置被选择，实际的变量名称是随意的，与位置无关。任何没有在解构模式中明确指定的项都会被忽略。记住，数组本身并没有发生任何改变。

你也可以在解构模式中忽略一些项，只给所需的项提供变量名。例如，若只想获取数组中的第三个元素，那么不必给前两项提供变量名。如下所示：

```
let colors = [ "red", "green", "blue" ];

let [ , , thirdColor ] = colors;

console.log(thirdColor);        // "blue"
```

此代码使用了解构赋值来获取 `colors` 的第三个项。模式中 `thirdColor` 之前的逗号，是为数组前面的项提供的占位符。使用这种方法，你就可以轻易从数组任意位置取出值，而无须给其他项提供变量名。

> 与对象解构相似，在使用 `var` 、 `let` 、 `const` 进行数组解构时，必须提供初始化器。

解构赋值

你可以在赋值表达式中使用数组解构，但是与对象解构不同，不必将表达式包含在圆括号内，例如：

```
let colors = [ "red", "green", "blue" ],
    firstColor = "black",
    secondColor = "purple";

[ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

此代码中解构赋值的工作方式与上例相似，唯一区别是 `firstColor` 与 `secondColor` 变量已经被声明过了。大多数情况下，对数组解构赋值仅需了解以上知识，但还有一个实用的小技巧。

数组解构赋值有一个非常独特的用法，能轻易地互换两个变量的值。互换变量值在排序算法中十分常用，而在 ES5 中需要使用第三个变量作为临时变量，正如下例：

```
// 在 ES5 中互换值
let a = 1,
    b = 2,
    tmp;

tmp = a;
a = b;
b = tmp;

console.log(a);     // 2
console.log(b);     // 1
```

其中的 `tmp` 变量对于互换 `a` 与 `b` 的值来说是必要的。然而若使用数组解构赋值，就不再需要这个额外变量。以下演示了在 ES6 中如何互换变量值：

```
// 在 ES6 中互换值
let a = 1,
    b = 2;

[ a, b ] = [ b, a ];

console.log(a);     // 2
console.log(b);     // 1
```

本例中的数组解构赋值看起来如同镜像。赋值语句左侧（即等号之前）的解构模式如同其他数组解构的范例，右侧则是为了互换而临时创建的数组字面量。 `b` 与 `a` 的值分别被复制到临时数组的第一个与第二个位置，对该数组进行解构，于是两个变量就互换了它们的值。

> 与对象解构赋值相同，若等号右侧的计算结果为 `null` 或 `undefined` ，那么数组解构赋值表达式也会抛出错误。

默认值

数组解构赋值同样允许在数组任意位置指定默认值。当指定位置的项不存在、或其值为 `undefined` ，那么该默认值就会被使用。例如：

```
let colors = [ "red" ];

let [ firstColor, secondColor = "green" ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

此代码的 `colors` 数组只有一个项，因此没有能与 `secondColor` 匹配的项，又由于此处有个默认值， `secondColor` 的值就被设置为 `"green"` 而非 `undefined` 。

嵌套的解构

与解构嵌套的对象相似，可以用类似的方式来解构嵌套的数组。在整个解构模式中插入另一个数组模式，解构操作就会下探到嵌套的数组中，就像这样：

```
let colors = [ "red", [ "green", "lightgreen" ], "blue" ];

// 随后

let [ firstColor, [ secondColor ] ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

此处的 `secondColor` 变量指向了 `colors` 数组中的 `"green"` 值，该项被包含在第二个数组中，因此解构模式就必须将 `secondColor` 包裹上方括号。与对象解构相似，你也能使用任意深度的数组嵌套。

剩余项

第三章介绍过函数的剩余参数，而数组解构有个类似的、名为剩余项（ **rest items** ）的概念，它使用 `...` 语法来将剩余的项目赋值给一个指定的变量，此处有个范例：

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, ...restColors ] = colors;

console.log(firstColor);          // "red"
console.log(restColors.length); // 2
console.log(restColors[0]);      // "green"
console.log(restColors[1]);      // "blue"
```

`colors` 数组的第一项被赋值给了 `firstColor` 变量，而剩余的则赋值给了一个新的 `restColors` 数组； `restColors` 数组有两个项： `"green"` 与 `"blue"` 。若要在取出特定项之余保留剩余的值，则剩余项就会非常有用，但它还有另一个有用的功能。

在 JS 中，方便地克隆数组是个明显被遗漏的功能。在 ES5 中开发者往往使用一个简单的方式来克隆数组，也就是用 `concat()` 方法，例如：

```
// 在 ES5 中克隆数组
var colors = [ "red", "green", "blue" ];
var clonedColors = colors.concat();

console.log(clonedColors);      //"[red,green,blue]"
```

尽管 `concat()` 方法的本意是合并两个数组，但若不使用任何参数来调用此方法，就会获得原数组的一个克隆品。而在 ES6 中，你可以使用剩余项的语法来达到同样效果。实现如下：

```
// 在 ES6 中克隆数组
let colors = [ "red", "green", "blue" ];
let [ ...clonedColors ] = colors;

console.log(clonedColors);      //"[red,green,blue]"
```

在本例中，剩余项被用于将 `colors` 数组的值复制到 `clonedColors` 数组中。虽然从表面上来看，这么做未必能让开发者复制数组的意图体现得比使用 `concat()` 方法更明显，但这依然是个值得关注的技巧。

剩余项必须是数组解构模式中最后的部分，之后不能再有逗号，否则就会有语法错误。

混合解构

对象与数组解构能被组合使用，以创建更复杂的解构表达式。在对象与数组混合而成的结构中，能在其中准确提取所需的信息片段。例如：

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        },
        range: [0, 3]
    };

let {
    loc: { start },
    range: [ startIndex ]
} = node;

console.log(start.line);        // 1
console.log(start.column);      // 1
console.log(startIndex);        // 0
```

此代码将 `node.loc.start` 与 `node.range[0]` 提取出来，并将它们的值分别存储到 `start` 与 `startIndex` 上。要记住解构模式中的 `loc:` 与 `range:` 只是对应于 `node` 对象中属性的位置。混合使用对象与数组解构，`node` 的任何部分都能提取出来。对于从 JOSN 配置结构中抽取数据来说，这种方法尤其有用，因为它不需要探索整个结构。

参数解构

解构还有一个特别有用的场景，即在传递函数参数时。当 JS 的函数接收大量可选参数时，一种常用模式是创建一个包含附加参数的 `options` 对象，就像这样：

```
// options 上的属性表示附加参数
function setCookie(name, value, options) {

    options = options || {};

    let secure = options.secure,
        path = options.path,
        domain = options.domain,
        expires = options.expires;

    // 设置 cookie 的代码
}

// 第三个参数映射到 options
setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

很多 JS 库都包含了与此例 `setCookie()` 类似的函数。在此函数内， `name` 与 `value` 参数是必需的，而 `secure` 、 `path` 、 `domain` 与 `expires` 则可选。并且由于此处对其余数据并没有顺序要求，将它们作为 `options` 对象的特定属性会更有效，避免列出一堆额外的具名参数。虽然这种方法有用，但无法仅通过查看函数定义就判断出函数所期望的输入，你必须阅读函数体的代码。

参数解构提供了更清楚标明函数期望输入的替代方案。它使用对象或数组解构的模式替代了具名参数。请查看下例中重写版本的 `setCookie()` 函数以了解实际效果：

```
function setCookie(name, value, { secure, path, domain, expires }) {

    // 设置 cookie 的代码
}

setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

此函数的行为类似上例，但此时第三个参数使用了解构来抽取必要的数据。现在对于 `setCookie()` 函数的使用者来说，解构参数之外的参数明显是必需的；而可选项目存在于额外的参数对象中，这同样显而易见；同时，若使用了第三个参数，其中应当包含什么值也相当明确。解构参数在未传递值的情况下会被设为 `undefined` ，类似于常规参数。

> 参数解构拥有此前你在本章已经学过的其他解构方式的所有能力。可以在其中使用默认参数、混合解构，或为变量重命名。

解构参数不可缺失

参数解构有一个怪异之处，在默认情况下，调用函数时未给参数解构传值会抛出错误。例如，用以下方式调用上例中的 `setCookie()` 函数就会出错：

```
// 出错！
setCookie("type", "js");
```

调用时第三个参数缺失了，它的值就变成了 `undefined` 。这导致了一个错误，因为参数解构实际上只是解构声明的简写。当 `setCookie()` 函数被调用时，JS 引擎实际上是这么做的：

```
function setCookie(name, value, options) {

    let { secure, path, domain, expires } = options;

    // 设置 cookie 的代码
}
```

既然解构在右侧的值为 `null` 或 `undefined` 时会抛出错误，那么未向 `setCookie()` 函数传递第三个参数也就同样会出错。

若解构的参数是必选参数，那么上述特性并不会令人困扰。但若期望它是可选的，就需要给解构的参数提供默认值来解决问题，就像这样：

```
function setCookie(name, value, { secure, path, domain, expires } = {}) {

    // ...
}
```

此例为第三个参数提供了一个空对象作为其默认值，也就意味着若未向 `setCookie()` 函数传递第三个参数，则 `secure` 、 `path` 、 `domain` 与 `expires` 的值全都会是 `undefined` ，此时不会有错误被抛出。

参数解构的默认值

你可以为参数解构提供可解构的默认值，就像在解构赋值时所做的那样，只需在其中每个参数后面添加等号并指定默认值即可。例如：

```
function setCookie(name, value,
    {
        secure = false,
        path = "/",
        domain = "example.com",
        expires = new Date(Date.now() + 360000000)
    } = {}
) {

    // ...
}
```

此代码中参数解构给每个属性都提供了默认值，所以你可以避免检查指定属性是否已被传入。而整个解构的参数同样有一个默认值，即一个空对象，令该参数成为可选参数。这么做使得函数声明看起来比平时要复杂一些，但只是为确保每个参数都有可用的值而付出的微小代价。

> 译注：
>
> 上面的参数解构只有一个缺点，也就是当传入参数值为 `null` 时会引发程序异常。只有 `null` 与 `undefined` 是无法被解构的，而传入 `undefined` 会触发默认参数的使用条件，从而避免了异常；但传入 `null` 就不会有这么幸运了，它既不会触发默认参数，也不能被解构，从而导致异常。

总结

解构让 JS 操作对象与数组变得更容易。使用熟悉的对象字面量与数组字面量语法，可以将数据结构分离并只获取所需信息。对象解构模式允许你从对象中进行提取，而数组模式则能用于数组。

对象与数组解构都能在属性或项未定义时为其提供默认值；在赋值表达式右侧的值为 `null` 或 `undefined` 时，两种模式都会抛出错误。你也可以在深层嵌套的数据结构中使用对象与数组解构，下探到该结构的任意深度。

使用 `var` 、 `let` 或 `const` 的解构声明来创建变量，就必须提供初始化器。解构赋值能替代其他赋值，并且允许解构到对象属性或已有变量上。

参数解构使用解构语法作为函数的参数，让“选项”（ options ）对象更加清晰透明，实际所需数据可以与具名参数一并列出。解构的参数可以是对象模式、数组模式或混合模式，并能使用解构的所有特性。

第六章 符号与符号属性

在 JS 已有的基本类型（字符串、数值、布尔、 `null` 与 `undefined` ）之外， ES6 引入了一种新的基本类型：符号（ Symbol ）。 符号起初被设计用于创建对象私有成员，而这也是 JS 开发者期待已久的特性。在符号诞生之前，将字符串作为属性名称导致属性可以被轻易访问，无论使用何种命名规则。而“私有名称”意味着开发者可以创建非字符串类型的属性名称，由此可以防止使用常规手段来探查这些名称。

“私有名称”提案最终发展成为 ES6 中的符号，而本章将会教你如何有效使用它。虽然它只保留了实现细节（即：引入了非字符串类型的属性名）而丢弃了私有性意图，但它仍然显著有别于对象的其余属性。

创建符号值

符号没有字面量形式，这在 JS 的基本类型中是独一无二的，有别于布尔类型的 `true` 或数值类型的 `42` 等等。你可以使用全局 `Symbol` 函数来创建一个符号值，正如下面这个例子：

```
let firstName = Symbol();
let person = {};

person[firstName] = "Nicholas";
console.log(person[firstName]);     // "Nicholas"
```

此代码创建了一个符号类型的 `firstName` 变量，并将它作为 `person` 对象的一个属性，而每次访问该属性都要使用这个符号值。为符号变量适当命名是个好主意，这样便能轻易说明它的用意。

> 由于符号值是基本类型的值，因此调用 `new Symbol()` 将会抛出错误。你可以通过 `new Object(yourSymbol)` 来创建一个符号实例，但尚不清楚这能有什么作用。

`Symbol` 函数还可以接受一个额外的参数用于描述符号值，该描述并不能用来访问对应属性，但它能用于调试，例如：

```
let firstName = Symbol("first name");
let person = {};

person[firstName] = "Nicholas";

console.log("first name" in person);        // false
console.log(person[firstName]);             // "Nicholas"
console.log(firstName);                     // "Symbol(first name)"
```

符号的描述信息被存储在内部属性 `[[Description]]` 中，当符号的 `toString()` 方法被显式或隐式调用时，该属性都会被读取。在本例中， `console.log()` 隐式调用了 `firstName` 变量的 `toString()` 方法，于是描述信息就被输出到日志。此外没有任何办法可以从代码中直接访问 `[[Description]]` 属性。我建议始终应给符号提供描述信息，以便更好地阅读代码或进行调试。

> 识别符号值
>
> 由于符号是基本类型的值，因此你可以使用 `typeof` 运算符来判断一个变量是否为符号。 ES6 扩充了 `typeof` 的功能以便让它能返回 `"symbol"` ，例如：
>
> ```
> let symbol = Symbol("test symbol");
> console.log(typeof symbol);         // "symbol"
> ```
>
> 尽管有其他办法可以判断一个变量是否为符号， `typeof` 运算符依然是最准确、最优先的判别手段。

使用符号值

你可以在任意能使用“可计算属性名”的场合使用符号。此前的例子已经展示了符号的方括号用法，而在对象的“可计算字面量属性名”中也能使用符号，还能在 `Object.defineProperty()` 或 `Object.defineProperties()` 调用中使用它，例如：

```
let firstName = Symbol("first name");

// 使用一个可计算字面量属性
let person = {
    [firstName]: "Nicholas"
};

// 让该属性变为只读
Object.defineProperty(person, firstName, { writable: false });

let lastName = Symbol("last name");

Object.defineProperties(person, {
    [lastName]: {
        value: "Zakas",
        writable: false
    }
});

console.log(person[firstName]);     // "Nicholas"
console.log(person[lastName]);      // "Zakas"
```

这个例子首先使用对象的“可计算字面量属性”创建了一个符号类型的属性 `firstName` ，该属性是可枚举的（译注：后面会提到与普通属性的差异）。下一行代码将该属性设置为只读。接下来，使用 `Object.defineProperties()` 方法创建了一个只读的符号类型属性 `lastName` ，而此时再次使用了“可计算字面量属性”方式，并将其加入第二个调用参数。

既然能在任意可使用“可计算属性名”的场合使用符号，你就需要一种在不同代码段中共享符号值的体系，以便更有效地使用它们。

共享符号值

你或许想在不同的代码段中使用相同的符号值，例如：假设在应用中需要在两个不同的对象类型中使用同一个符号属性，用于表示一个唯一标识符。跨越文件或代码来追踪符号值既困难又易错，为此， ES6 提供了“全局符号注册表”供你随时访问。

若你想创建共享符号值，应使用 `Symbol.for()` 方法而不是 `Symbol()` 方法。 `Symbol.for()` 方法仅接受字符串类型的单个参数，作为目标符号值的标识符，此参数同时也会成为该符号的描述信息。例如：

```
let uid = Symbol.for("uid");
let object = {};

object[uid] = "12345";

console.log(object[uid]);       // "12345"
console.log(uid);               // "Symbol(uid)"
```

`Symbol.for()` 方法首先会搜索全局符号注册表，看是否存在一个键值为 `"uid"` 的符号值。若是，该方法会返回这个已存在的符号值；否则，会创建一个新的符号值，并使用该键值将其记录到全局符号注册表中，然后返回这个新的符号值。这就意味着此后使用同一个键值去调用 `Symbol.for()` 方法都会返回同一个符号值，就像下面这个例子：

```
let uid = Symbol.for("uid");
let object = {
    [uid]: "12345"
};

console.log(object[uid]);       // "12345"
console.log(uid);               // "Symbol(uid)"

let uid2 = Symbol.for("uid");

console.log(uid === uid2);      // true
console.log(object[uid2]);      // "12345"
console.log(uid2);              // "Symbol(uid)"
```

本例中， `uid` 与 `uid2` 包含同一个符号值，因此它们可以互换使用。第一次调用 `Symbol.for()` 创建了这个符号值，而第二次调用则从全局符号注册表中将其检索了出来。

共享符号值还有另一个独特用法，你可以使用 `Symbol.keyFor()` 方法在全局符号注册表中根据符号值检索出对应的键值，例如：

```
let uid = Symbol.for("uid");
console.log(Symbol.keyFor(uid));    // "uid"

let uid2 = Symbol.for("uid");
console.log(Symbol.keyFor(uid2));   // "uid"

let uid3 = Symbol("uid");
console.log(Symbol.keyFor(uid3));   // undefined
```

注意：使用符号值 `uid` 与 `uid2` 都返回了键值 `"uid"` ，而符号值 `uid3` 在全局符号注册表中并不存在，因此没有关联的键值， `Symbol.keyFor()` 方法只会返回 `undefined` 。

> 全局符号注册表类似于全局作用域，是一个共享环境，这意味着你不应当假设其中是否已存在某些值。在使用第三方组件时，为符号的键值使用命名空间能够减少命名冲突的可能性，举个例子： jQuery 代码应当为它的所有键值使用 `"jquery."` 的前缀，如 `"jquery.element"` 或类似的形式。

符号值的转换

类型转换是 JS 语言重要的一部分，能够非常灵活地将一种数据类型转换到其他类型。然而符号类型在进行转换时非常不灵活，因为其他类型缺乏与符号值的合理等价，尤其是符号值无法被转换为字符串值或数值。因此将符号作为属性所达成的效果，是其他类型所无法替代的。

本章之前的例子使用了 `console.log()` 来展示符号值的输出，这样做自动调用了符号的 `String()` 方法来产生输出。你也能直接调用 `String()` 方法来获取相同结果，例如：

```
let uid = Symbol.for("uid"),
    desc = String(uid);

console.log(desc);              // "Symbol(uid)"
```

`String()` 方法调用了 `uid.toString()` 来获取符号的字符串描述信息。但若你想将符号与字符串直接拼接，则会引发错误：

```
let uid = Symbol.for("uid"),
    desc = uid + "";            // 引发错误！
```

将 `uid` 与空字符串相连接，会首先要求把 `uid` 转换为一个字符串，而这会引发错误，从而中止了转换行为。

类似地，你不能将符号转换为数值，对符号使用所有数学运算符都会引发错误，例如：

```
let uid = Symbol.for("uid"),
    sum = uid / 1;            // 引发错误！
```

此例试图把符号值除以 1 ，同样引发了错误。无论对符号使用哪种数学运算符都会导致错误，不过使用逻辑运算符就不会，因为符号值在逻辑运算中会被认为等价于 `true` ，就像 JS 中其他的非空值那样。

检索符号属性

`Object.keys()` 与 `Object.getOwnPropertyNames()` 方法可以检索对象的所有属性名称，前者返回所有的可枚举属性名称，而后者的返回值则不会顾虑属性是否可枚举。然而为了延续它们在 ES5 中的功能，二者都不能返回符号类型的属性。 ES6 新增了 `Object.getOwnPropertySymbols()` 方法，以便检索对象的符号类型属性。

> 译注：前面提到符号属性默认是可枚举的，使用 `getOwnPropertyDescriptor()` 方法去查看即可证实这一点。原作误认为符号属性默认不可枚举，后来作了修正。其原因就在于 `Object.keys()` 是忽略符号属性的， for-in 循环遍历也受到类似限制，这些特性会对判断符号属性是否可枚举造成干扰。

`Object.getOwnPropertySymbols()` 方法会返回一个数组，包含了对象自有属性名中的符号值，例如：

```
let uid = Symbol.for("uid");
let object = {
    [uid]: "12345"
};

let symbols = Object.getOwnPropertySymbols(object);

console.log(symbols.length);        // 1
console.log(symbols[0]);            // "Symbol(uid)"
console.log(object[symbols[0]]);    // "12345"
```

这段代码中， `object` 对象只拥有一个名为 `uid` 的符号类型属性， `Object.getOwnPropertySymbols()` 方法返回的数组包含了它。

所有对象起初都不包含任何自有符号类型属性，但对象可以从它们的原型上继承符号类型属性。 ES6 预定义了一些此类属性，它们被称为“知名符号”。

使用知名符号暴露内部方法

ES5 的中心主题之一是定义并披露了一些魔术般的成分，而这些功能在当时无法由开发者自行模拟。 ES6 延续了这些工作，进一步暴露了原先属于语言内部逻辑的部分，从而允许使用符号类型的原型属性来定义某些对象的基础行为。

ES6 定义了“知名符号”来代表 JS 中一些公共行为，而这些行为此前被认为只能是内部操作。每一个知名符号都对应全局 `Symbol` 对象的一个属性，例如 `Symbol.create` 。

这些知名符号是：

- `Symbol.hasInstance` ：供 `instanceof` 运算符使用的一个方法，用于判断对象继承关系。
- `Symbol.isConcatSpreadable` ：一个布尔类型值，在集合对象作为参数传递给 `Array.prototype.concat()` 方法时，指示是否要将该集合的元素扁平化。

- `Symbol.iterator` ：返回迭代器（参阅第八章）的一个方法。
- `Symbol.match` ：供 `String.prototype.match()` 函数使用的一个方法，用于比较字符串。
- `Symbol.replace` ：供 `String.prototype.replace()` 函数使用的一个方法，用于替换子字符串。
- `Symbol.search` ：供 `String.prototype.search()` 函数使用的一个方法，用于定位子字符串。
- `Symbol.species` ：用于产生派生对象（参阅第九章）的构造器。
- `Symbol.split` ：供 `String.prototype.split()` 函数使用的一个方法，用于分割字符串。
- `Symbol.toPrimitive` ：返回对象所对应的基本类型值的一个方法。
- `Symbol.toStringTag` ：供 `String.prototype.toString()` 函数使用的一个方法，用于创建对象的描述信息。
- `Symbol.unscopables` ：一个对象，其属性指示了哪些属性名不允许被包含在 `with` 语句中。

一些公用的知名符号将在下面诸小节进行介绍，而其余的则会在本书其他部分中讨论，以保证它们出现在正确的上下文中。

> 重写知名符号所定义的方法，会把一个普通对象改变成奇异对象，因为它改变了一些默认的内部行为。这并不会对你的代码造成实质影响，它只是改变了规范描述对象的方式。

Symbol.hasInstance 属性

每个函数都具有一个 `Symbol.hasInstance` 方法，用于判断指定对象是否为本函数的一个实例。这个方法定义在 `Function.prototype` 上，因此所有函数都继承了面对 `instanceof` 运算符时的默认行为。 `Symbol.hasInstance` 属性自身是不可写入、不可配置、不可枚举的，从而保证它不会被错误重写。

`Symbol.hasInstance` 方法只接受单个参数，即需要检测的值。如果该值是本函数的一个实例，则方法会返回 `true` 。为了理解该方法是如何工作的，可研究下述代码：

```
obj instanceof Array;
```

这句代码等价于：

```
Array[Symbol.hasInstance](obj);
```

ES6 从本质上将 `instanceof` 运算符重定义为上述方法调用的简写语法，这样使用 `instanceof` 便会触发一次方法调用，实际上允许你改变该运算符的工作方式。

例如，假设你想定义一个函数，使得任意对象都不会被判断为该函数的一个实例，你可以采用硬编码的方式让该函数的 `Symbol.hasInstance` 方法始终返回 `false` ，就像这样：

```
function MyObject() {
    // ...
}

Object.defineProperty(MyObject, Symbol.hasInstance, {
    value: function(v) {
        return false;
    }
});

let obj = new MyObject();

console.log(obj instanceof MyObject);       // false
```

要重写一个不可写入的属性，你必须像这个例子一样使用 `Object.defineProperty()` 。此代码将 `Symbol.hasInstance` 方法重写为一个始终返回 `false` 的函数，所以此后即使传入的对象确实是 `MyObject` 类的一个实例， `instanceof` 运算符仍然会返回 `false` 。

当然，你可以基于任意条件来判断一个值是否为实例。例如，将介于 1 到 100 之间的数值认定为一个特殊的数值类型，为此你可以书写如下代码：

```
function SpecialNumber() {
    // empty
}

Object.defineProperty(SpecialNumber, Symbol.hasInstance, {
    value: function(v) {
        return (v instanceof Number) && (v >=1 && v <= 100);
    }
});

let two = new Number(2),
    zero = new Number(0);

console.log(two instanceof SpecialNumber);    // true
console.log(zero instanceof SpecialNumber);   // false
```

此代码重写了 `Symbol.hasInstance` 方法，在目标对象是数值对象的实例、并且其值介于 1 到 100 之间时，返回 `true` 。于是， `SpecialNumber` 类会把变量 `two` 判断为自身的一个实例，即使二者之间并不存在直接关联。需要注意的是： `instanceof` 的左操作数必须是一个对象，以便触发 `Symbol.hasInstance` 调用；否则 `instanceof` 只会直接返回 `false` 。

> 你可以重写所有内置函数（例如 `Date` 或 `Error` ）的 `Symbol.hasInstance` 属性，但我并不建议这么做，因为这会让你的代码变得既难以预测又混乱。最好仅在必要时对你自己的函数重写 `Symbol.hasInstance` 。

Symbol.isConcatSpreadable

JS 在数组上设计了 `concat()` 方法用于将两个数组连接到一起，此处示范了用法：

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ]);

console.log(colors2.length);    // 4
console.log(colors2);           // ["red","green","blue","black"]
```

此代码将一个新数组连接到 `colors1` 末尾，并创建了 `colors2` ，后者包含了前两个数组中所有的项。不过， `concat()` 方法也可以接受非数组的参数，此时这些参数只会直接被添加到数组末尾，例如：

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ], "brown");

console.log(colors2.length);    // 5
console.log(colors2);           // ["red","green","blue","black","brown"]
```

此代码向 `concat()` 方法传递了一个额外参数 `"brown"` ，使得它成为数组 `colors2` 的第 5 项。为何数组类型的参数与字符串类型的参数会被区别对待？这是因为 JS 规范要求此时数组类型的参数需要被自动分离出各个子项，而其他类型的参数无需如此处理。在 ES6 之前，没有任何手段可以改变这种行为。

`Symbol.isConcatSpreadable` 属性是一个布尔类型的属性，它表示目标对象拥有长度属性与数值类型的键、并且数值类型键所对应的属性值在参与 `concat()` 调用时需要被分离为个体。该符号与其他的知名符号不同，默认情况下并不会作为任意常规对象的属性。它只出现在特定类型的对象上，用来标示该对象在作为 `concat()` 参数时应如何工作，从而有效改变该对象的默认行为。你可以用它来定义任意类型的对象，让该对象在参与 `concat()` 调用时能与数组类似，例如：

```
let collection = {
    0: "Hello",
    1: "world",
    length: 2,
    [Symbol.isConcatSpreadable]: true
};

let messages = [ "Hi" ].concat(collection);

console.log(messages.length);    // 3
console.log(messages);           // ["hi","Hello","world"]
```

本例中的 `collection` 对象的特征类似于数组：拥有长度属性以及两个数值类型的键，并且 `Symbol.isConcatSpreadable` 属性值被设为 `true` ，用于指示该对象在被添加到数组时应该使用分离的属性值。当 `collection` 对象被传递给 `concat()` 方法时， `"Hello"` 与 `"world"` 被分离为独立的项，并跟在 `"hi"` 元素之后。

> 你也可以将数组的子类的 `Symbol.isConcatSpreadable` 属性值设为 `false` ，以便在 `concat()` 调用时避免项目被分离。子类的介绍位于第九章。

Symbol.match 、 Symbol.replace 、 Symbol.search 与 Symbol.split

在 JS 中，字符串与正则表达式有着密切的联系，尤其是字符串具有几个可以接受正则表达式作为参数的方法：

- `match(regex)` ：判断指定字符串是否与一个正则表达式相匹配；
- `replace(regex, replacement)` ：对正则表达式的匹配结果进行替换；
- `search(regex)` ：在字符串内对正则表达式的匹配结果进行定位；
- `split(regex)` ：使用正则表达式将字符串分割为数组。

这些与正则表达式交互的方法，在 ES6 之前的实现细节是对开发者隐藏的，使得开发者无法将自定义对象模拟成正则表达式（并将它们传递给字符串的这些方法）。而 ES6 定义了 4 个符号以及对应的方法，将内置的 `RegExp` 对象上的原生行为外包出来。

这 4 个符号表示将正则表达式作为字符串对应方法的第一个参数传入时应调用的方法， `Symbol.match` 对应 `match()` 方法， `Symbol.replace` 对应 `replace()` ， `Symbol.search` 对应 `search()` ， `Symbol.split` 则对应 `split()` 。这些符号属性被定义在 `RegExp.prototype` 上作为默认实现，以供对应的字符串方法使用。

了解这些之后，你就可以创建一个类似于正则表达式的对象，以便配合字符串的那些方法使用。在代码中使用下述的符号函数即可：

- `Symbol.match` ：此函数接受一个字符串参数，并返回一个包含匹配结果的数组；若匹配失败，则返回 `null` 。
- `Symbol.replace` ：此函数接受一个字符串参数与一个替换用的字符串，并返回替换后的结果字符串。
- `Symbol.search` ：此函数接受一个字符串参数，并返回匹配结果的数值索引；若匹配失败，则返回 -1。
- `Symbol.split` ：此函数接受一个字符串参数，并返回一个用匹配值分割而成的字符串数组。

在对象上定义这些属性，允许你创建不使用正则表达式却能进行模式匹配的对象，并且允许在任何方法中替代正则表达式。此处有个例子，展示了这些符号的用法：

```
// 有效等价于 /^.{10}$/
let hasLengthOf10 = {
    [Symbol.match]: function(value) {
        return value.length === 10 ? [value] : null;
    },
    [Symbol.replace]: function(value, replacement) {
        return value.length === 10 ? replacement : value;
    },
    [Symbol.search]: function(value) {
        return value.length === 10 ? 0 : -1;
    },
    [Symbol.split]: function(value) {
        return value.length === 10 ? ["", ""] : [value];
    }
};

let message1 = "Hello world",   // 11 characters
    message2 = "Hello John";    // 10 characters


let match1 = message1.match(hasLengthOf10),
    match2 = message2.match(hasLengthOf10);

console.log(match1);            // null
console.log(match2);            // ["Hello John"]

let replace1 = message1.replace(hasLengthOf10, "Howdy!"),
    replace2 = message2.replace(hasLengthOf10, "Howdy!");

console.log(replace1);          // "Hello world"
console.log(replace2);          // "Howdy!"

let search1 = message1.search(hasLengthOf10),
    search2 = message2.search(hasLengthOf10);

console.log(search1);           // -1
console.log(search2);           // 0

let split1 = message1.split(hasLengthOf10),
    split2 = message2.split(hasLengthOf10);

console.log(split1);            // ["Hello world"]
console.log(split2);            // ["", ""]
```

`hasLengthOf10` 对象模拟了正则表达式的工作方式，在字符串长度恰好为 10 的时候起作用。`hasLengthOf10` 对象上的四个方法都对相应的符号属性进行了实现，并依次在两个字符串上被调用。第一个字符串 `message1` 长度为 11 ，因此不会匹配成功；而字符串 `message2` 长度为10，可以正确匹配。尽管 `hasLengthOf10` 对象不是正则表达式，但它仍然作为参数传递给这些字符串方法，并能够正常工作。

这虽然仅是一个简单例子，却表明能够进行比现有正则表达式更复杂的匹配，让自定义模式匹配更加可行。

Symbol.toPrimitive

JS 经常在使用特定运算符的时候试图进行隐式转换，以便将对象转换为基本类型值。例如，当你使用相等（ `==` ）运算符来对字符串与对象进行比较的时候，该对象会在比较之前被转换为一个基本类型值。到底转换为何种基本类型值，在此前属于内部操作，而 ES6 则通过 `Symbol.toPrimitive` 方法将其暴露出来，以便让其可被更改。

`Symbol.toPrimitive` 方法被定义在所有常规类型的原型上，规定了在对象被转换为基本类型值的时候会发生什么。当需要转换时， `Symbol.toPrimitive` 会被调用，并按照规范传入一个提示性的字符串参数。该参数有 3 种可能：当参数值为 `"number"` 的时候， `Symbol.toPrimitive` 应当返回一个数值；当参数值为 `"string"` 的时候，应当返回一个字符串；而当参数为 `"default"` 的时候，对返回值类型没有特别要求。

对于大部分常规对象，“数值模式”依次会有下述行为：

1. 调用 `valueOf()` 方法，若结果是一个基本类型值，那么返回它；
2. 否则，调用 `toString()` 方法，若结果是一个基本类型值，那么返回它；
3. 否则，抛出一个错误。

类似的，对于大部分常规对象，“字符串模式”依次会有下述行为：

1. 调用 `toString()` 方法，若结果是一个基本类型值，那么返回它；
2. 否则，调用 `valueOf()` 方法，若结果是一个基本类型值，那么返回它；
3. 否则，抛出一个错误。

在多数情况下，常规对象的默认模式都等价于数值模式（只有 `Date` 类型例外，它默认使用字符串模式）。通过定义 `Symbol.toPrimitive` 方法，你可以重写这些默认的转换行为。

> “默认模式”只在使用 == 运算符、 + 运算符、或者传递单一参数给 `Date` 构造器的时候被使用，而大部分运算符都使用字符串模式或是数值模式。

使用 `Symbol.toPrimitive` 属性并将一个函数赋值给它，便可以重写默认的转换行为，例如：

```
function Temperature(degrees) {
    this.degrees = degrees;
}

Temperature.prototype[Symbol.toPrimitive] = function(hint) {

    switch (hint) {
        case "string":
            return this.degrees + "\u00b0"; // 温度符号

        case "number":
            return this.degrees;

        case "default":
            return this.degrees + " degrees";
    }
};

let freezing = new Temperature(32);

console.log(freezing + "!");            // "32 degrees!"
console.log(freezing / 2);              // 16
console.log(String(freezing));          // "32°"
```

这段脚本定义了一个 `Temperature` 构造器，并重写了其原型上的 `Symbol.toPrimitive` 方法。返回值会依据方法的提示性参数而有所不同，可以使用字符串模式、数值模式或是默认模式，而该提示性参数会在调用时由 JS 引擎自动填写。字符串模式中， `Temperature` 函数返回的温度会附带着 Unicode 温度符号；数值模式只会返回温度数字；而默认模式中，返回的温度会附带着字符串后缀 `"degrees"` 。

此后的三个 log 语句分别触发了不同的提示性参数值： `+` 运算符使用 `"default"` 触发了默认模式； `/` 运算符使用 `"number"` 触发了数值模式；而 `String()` 函数则使用了 `"string"` 触发了字符串模式。允许在三种模式下返回互不相同的结果，但一般来说默认模式的返回值都会与字符串模式或数值模式相等。

Symbol.toStringTag

JS 最有趣的课题之一是在多个不同的全局执行环境中使用，这种情况会在浏览器页面包含内联帧（ iframe ）时出现，此时页面与内联帧均拥有各自的全局执行环境。大多数情况下这并不是一个问题，使用一些轻量级的转换操作就能够在不同的运行环境之间传递数据。而当对象已在环境之间经历了传递，再要识别它们的类型时，问题就来了。

该问题的典型例子就是从内联帧向容器页面传递数组，或者反过来。在 ES6 术语中，内联帧与包含它的容器页面分别拥有一个不同的“域”，以作为 JS 的运行环境，每个“域”都拥有各自的全局作用域以及各自的全局对象拷贝。无论哪个“域”创建的数组都是正规的数组，但当它跨

域进行传递时，使用 `instanceof Array` 进行检测却会得到 `false` 的结果，因为该数组是由另外一个“域”的数组构造器创建的，不同于当前“域”的数组构造器。

识别问题的变通解决方法

面对识别数组这类问题，开发者迅速找到了一个好办法，他们发现通过调用常规的 `toString()` 方法，就会得到一个可预期的字符串结果。因此，很多 JS 库都包含了如下函数：

```
function isArray(value) {
    return Object.prototype.toString.call(value) === "[object Array]";
}

console.log(isArray([]));   // true
```

这看起来是一种迂回方式，但它在任何浏览器中都能非常准确地识别数组。在数组对象上调用 `toString()` 方法没什么用处，因为它会返回由数组元素拼接成的字符串；然而若在 `Object.prototype` 上调用 `toString()` 方法，却恰巧能达到目的：返回值会包含名为 `[[Class]]` 的内部定义名称。开发者可以在对象上使用这个方法，以获知 JS 引擎将该对象判断为什么类型。

开发者迅速意识到基于这种行为的不变性，可以用其来区别原生对象与开发者自建对象，其中最重要的范例就是 ES5 的 `JSON` 对象。

在 ES5 之前，许多开发者都使用了 Douglas Crockford 的 json2.js 脚本，用来创建全局的 `JSON` 对象。在浏览器开始实现 `JSON` 全局对象之后，就非常有必要区分全局 `JSON` 对象是 JS 运行环境自带的、还是由库文件引入的。使用与识别数组相同的技术，很多开发者创建了如下的函数：

```
function supportsNativeJSON() {
    return typeof JSON !== "undefined" &&
        Object.prototype.toString.call(JSON) === "[object JSON]";
}
```

`Object.prototype` 的特性允许开发者跨越内联帧界限去识别数组，而使用相同方式可以辨别 `JSON` 对象是否为原生的。非原生的 `JSON` 对象会返回 `[object Object]` ，而原生的 `JSON` 对象则会返回 `[object JSON]` 。这类方法也成为了识别原生对象的事实标准。

ES6 给出的答案

ES6 通过 `Symbol.toStringTag` 重定义了相关行为，该符号代表了所有对象的一个属性，定义了 `Object.prototype.toString.call()` 被调用时应当返回什么值。对于数组来说，在 `Symbol.toStringTag` 属性中存储了 `"Array"` 值，于是该函数的返回值也就是 `"Array"` 。

同样，你可以在自设对象上定义 `Symbol.toStringTag` 的值：

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Person";

let me = new Person("Nicholas");

console.log(me.toString());                         // "[object Person]"
console.log(Object.prototype.toString.call(me));    // "[object Person]"
```

本例在 `Person` 的原型上定义了 `Symbol.toStringTag` 属性，用于给它的字符串表现形式提供默认行为。由于 `Person` 的原型继承了 `Object.prototype.toString()` 方法，`Symbol.toStringTag` 的返回值在调用 `me.toString()` 的时候也会被使用。不过，你依然可以在该对象上定义你自己的 `toString()` 方法，让它有不同的返回值，而不用影响 `Object.prototype.toString.call()` 方法。此处有个例子：

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Person";

Person.prototype.toString = function() {
    return this.name;
};

let me = new Person("Nicholas");

console.log(me.toString());                         // "Nicholas"
console.log(Object.prototype.toString.call(me));    // "[object Person]"
```

这段代码让 `Object.prototype.toString.call()` 返回 `name` 属性的值。由于 `Person` 类的实例不再继承 `Object.prototype.toString()` 方法，调用 `me.toString()` 会显示不同的结果。

> 除非进行了特殊指定，否则所有对象都会从 `Object.prototype` 继承 `Symbol.toStringTag` 属性，其默认的属性值是字符串 `"Object"` 。

对于开发者自定义对象， `Symbol.toStringTag` 的返回值不受任何限制。例如，你可以自由使用 `"Array"` 作为 `Symbol.toStringTag` 属性的值，像这样：

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Array";

Person.prototype.toString = function() {
    return this.name;
};

let me = new Person("Nicholas");

console.log(me.toString());                         // "Nicholas"
console.log(Object.prototype.toString.call(me));    // "[object Array]"
```

在这段代码中，调用 `Object.prototype.toString()` 的结果是 `"[object Array]"` ，与在真实数组上调用的结果完全一样。这一点明确证实 `Object.prototype.toString()` 不再是用于识别对象类型的可靠方法。

改变原生对象的字符串标签也是可能的，只需要在对象的原型上对 `Symbol.toStringTag` 进行赋值，例如：

```
Array.prototype[Symbol.toStringTag] = "Magic";

let values = [];

console.log(Object.prototype.toString.call(values));    // "[object Magic]"
```

本例重写了数组的 `Symbol.toStringTag` 属性，导致 `Object.prototype.toString()` 被调用时会返回 `"[object Magic]"` 。尽管我建议不要用这种方式修改内置对象，但语言本身并没有禁止该行为。

Symbol.unscopables

`with` 语句是 JS 语言中最有争议的部分之一。它原本被设计用于减少重复代码的输入，但此后却遭受了全面的批评，因为它让代码变得更难理解，并且有负面性能影响，同时还易出错。

在严格模式下， `with` 语句最终被禁用了，而此限制同样影响了类与模块，因为它们默认启用并会被锁定在严格模式下。

尽管将来的代码无疑会停用 `with` 语句，但 ES6 仍然在非严格模式中提供了对于 `with` 语句的支持，以便向下兼容。为此需要寻找方法让使用 `with` 语句的代码能够继续适当工作。

为了理解这个任务的复杂性，可研究如下代码：

```
let values = [1, 2, 3],
    colors = ["red", "green", "blue"],
    color = "black";

with(colors) {
    push(color);
    push(...values);
}

console.log(colors);    // ["red", "green", "blue", "black", 1, 2, 3]
```

在此例中， `with` 语句内的两次 `push()` 调用等价于 `colors.push()` ，因为 `with` 语句为 `push` 添加了局部绑定； `color` 则引用了在 `with` 语句之外定义的变量；而 `values` 的本意也是如此。

但 ES6 为数组添加了一个 `values` 方法（可查阅第八章："迭代器与生成器"），这意味着在 ES6 的环境中， `with` 语句内部的 `values` 并不会指向局部变量 `values` ，而是会指向数组的 `values` 方法，从而会破坏代码的意图。 `Symbol.unscopables` 符号的出现正是为了解决这类问题。

`Symbol.unscopables` 符号在 `Array.prototype` 上使用，以指定哪些属性不允许在 `with` 语句内被绑定。 `Symbol.unscopables` 属性是一个对象，当提供该属性时，它的键就是用于忽略 `with` 语句绑定的标识符，键值为 `true` 代表屏蔽绑定。以下是数组 `Symbol.unscopables` 属性的默认值：

```
// 默认内置在 ES6 中
Array.prototype[Symbol.unscopables] = Object.assign(Object.create(null), {
    copyWithin: true,
    entries: true,
    fill: true,
    find: true,
    findIndex: true,
    keys: true,
    values: true
});
```

`Symbol.unscopables` 对象使用 `Object.create(null)` 创建，因此没有原型，并包含了 ES6 数组所有的新方法（可参阅第八章"迭代器与生成器"、第十章"数组"）。在 `with` 语句内不会再对这些方法进行绑定，因此旧代码可以继续工作而不会出问题。

一般而言，你无需在自定义的对象上设置 `Symbol.unscopables` 属性，除非使用了 `with` 语句、并修改了代码库中已有的对象。

总结

符号是 JS 新引入的基本类型值，它用于创建只有引用符号才能访问的属性。

虽然符号类型的属性不是眞正的私有属性，但它们难以被开发者无意修改，因此非常适用于需要提供一定层次保护的场合。

你可以为符号提供描述信息以便更容易地辨识它们的值。全局符号注册表允许你使用相同的描述信息，以便在不同的代码段中共享符号值，这样相同的符号值就可以在不同位置用于相同目的。

`Object.keys()` 或 `Object.getOwnPropertyNames()` 不会返回符号值，因此 ES6 新增了一个 `Object.getOwnPropertySymbols()` 方法，允许检索符号类型的对象属性。而你依然可以使用 `Object.defineProperty()` 与 `Object.defineProperties()` 方法对符号类型的属性进行修改。

“知名符号”使用了全局符号常量（例如 `Symbol.hasInstance` ），为常规对象定义了一些功能，而这些功能原先仅限内部使用。这些符号按规范使用 `Symbol.` 的前缀，允许开发者通过多种方式去修改常规对象的行为。

第七章 Set与Map

JS 长期以来都只存在一种集合类型，也就是数组类型（尽管有人会反对，声称所有非数组的对象都是键值对的集合，它们曾被用于与数组完全不同的用途）。数组在 JS 中的使用正如其他语言的数组一样，但缺少更多类型的集合导致数组也经常被当作队列与栈来使用。数组只使用了数值型的索引，而若需要有非数值型的索引，开发者便会使用非数组的对象，用它们来定制实现 Set 与 Map 。

Set 是不包含重复值的列表。一般不会像对待数组那样来访问 Set 中的某个项，更常见的操作是在 Set 中检查某个值是否存在。 **Map** 则是键与相对应的值的集合。因此， Map 中的每个项都存储了两块数据，通过指定所需读取的键即可检索对应的值。 Map 常被用作缓存，存储数据以便此后快速检索。由于 ES5 并未正式提供 Set 与 Map ，开发者就只能使用非数组的对象来模拟。

ES6 为 JS 添加了 Set 与 Map ，本章将论述这两种集合类型你所需了解的全部内容。

首先，我会论述在 ES6 之前开发者为了实现 Set 与 Map 而采用的变通方法，并描述这些方法的缺陷。在讲解这些重要的背景之后，我会介绍 Set 与 Map 在 ES6 中如何工作。

ES5 中的 Set 与 Map

在 ES5 中，开发者使用对象属性来模拟 Set 与 Map ，就像这样：

```
let set = Object.create(null);

set.foo = true;

// 检查属性的存在性
if (set.foo) {

    // 一些操作
}
```

本例中的 `set` 变量是一个原型为 `null` 的对象，确保在此对象上没有继承属性。使用对象的属性来检测唯一性，在 ES5 中是很常用的方法。当一个属性被添加到 `set` 对象时，它的值也被设为 `true` ，因此条件判断语句（例如本例中的 `if` 语句）就可以简单判断出该值是否存在。

使用对象模拟 Set 与模拟 Map 之间唯一实质区别是所存储的值。例如，以下例子将对象作为 Map 使用：

```
let map = Object.create(null);

map.foo = "bar";

// 提取一个值
let value = map.foo;

console.log(value);         // "bar"
```

此代码将字符串值 `"bar"` 存储在 `foo` 键上。与 Set 仅检查键的存在性不同， Map 多数被用来提取数据。

变通方法的问题

尽管在简单情况下将对象作为 Set 与 Map 来使用都是可行的，但一旦接触到对象属性的局限性，此方式就会碰壁。例如，由于对象属性的类型必须为字符串，你就必须保证任意两个键不能被转换为相同的字符串。研究以下代码：

```
let map = Object.create(null);

map[5] = "foo";

console.log(map["5"]);      // "foo"
```

本例将字符串值 `"foo"` 赋值到数值类型的键 `5` 上，而数值类型的键会在内部被转换为字符串，因此 `map["5"]` 与 `map[5]` 实际上引用了同一个属性。若需要将数值与字符串同时作为键来使用，这种内部转换就会引起问题。而若使用对象作为键，会出现另一个问题，例如：

```
let map = Object.create(null),
    key1 = {},
    key2 = {};

map[key1] = "foo";

console.log(map[key2]);     // "foo"
```

此处的 `map[key2]` 与 `map[key1]` 引用了同一个值。由于对象的属性只能是字符串， `key1` 与 `key2` 对象就必须被转换为字符串来使用，它们会被转换为同一个字符串，也就是对象默认的字符串表示 `"[object Object]"` ，从而指向了同一个属性。这种错误可能不太显眼，因为貌似合乎逻辑的假设是：如果在键上使用了不同对象，它们就应当指向不同的键。

由于对象会被转换为默认的字符串表现形式，它就难以被当作 Map 的键来使用，此问题同样存在于将对象作为 Set 来使用的尝试上。

当键的值为假值时， Map 也会遇到自身的特殊问题。在需要布尔值的位置（例如在 `if` 语句内），任何假值都会被自动转换为 false 。这种转换单独说来并不是问题，但需要对值的用法足够小心。例如，查看以下代码：

```
let map = Object.create(null);

map.count = 1;

// 是想检查 "count" 属性的存在性，还是想检查非零值？
if (map.count) {
    // ...
}
```

此例中 `map.count` 的用法存在歧义。此处的 `if` 语句是想检查 `map.count` 属性的存在性，还是想检查非零值？该 `if` 语句内的代码会被执行是因为 1 是真值。然而若 `map.count` 的值为 0 ，或者该属性不存在，则 `if` 语句内的代码都不会被执行。

在大型应用中，这类问题都是难以确认、难以调试的，这也是 ES6 新增 Set 与 Map 类型的首要原因。

> JS 存在 `in` 运算符，若属性存在于对象中，就会返回 `true` 而无须读取对象的属性值。不过， `in` 运算符会搜索对象的原型，这让它只有在处理原型为 `null` 的对象时才是安全的。但即使原型可靠，许多开发者仍然错误地使用上例那样的代码，而不使用 `in` 运算符。

ES6 的 Set

ES6 新增了 `Set` 类型，这是一种无重复值的有序列表。允许对 Set 包含的数据进行快速访问，从而能更有效地追踪离散值。

创建 Set 并添加项目

Set 使用 `new Set()` 来创建，而调用 `add()` 方法就能向 Set 中添加项目，检查 `size` 属性还能查看其中包含有多少项：

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.size);    // 2
```

Set 不会使用强制类型转换来判断值是否重复。这意味着 Set 可以同时包含数值 `5` 与 字符串 `"5"` ，将它们都作为相对独立的项（在 Set 内部的比较使用了第四章讨论过的 `Object.is()` 方法，来判断两个值是否相等，唯一的例外是 +0 与 -0 在 Set 中被判断为是相等的）。你还可以向 Set 添加多个对象，它们不会被合并为同一项：

```
let set = new Set(),
    key1 = {},
    key2 = {};

set.add(key1);
set.add(key2);

console.log(set.size);    // 2
```

由于 `key1` 与 `key2` 并不会被转换为字符串，所以它们在这个 Set 内部被认为是两个不同的项（记住：如果它们被转换为字符串，那么都会等于 `"[object Object]"` ）。

如果 `add()` 方法用相同值进行了多次调用，那么在第一次之后的调用实际上会被忽略：

```
let set = new Set();
set.add(5);
set.add("5");
set.add(5);     // 重复了，该调用被忽略

console.log(set.size);    // 2
```

你可以使用数组来初始化一个 Set ，并且 `Set` 构造器会确保不重复地使用这些值。例如：

```
let set = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
console.log(set.size);    // 5
```

在此例中，带有重复值的数组被用来初始化这个 Set 。虽然数值 5 在数组中出现了四次，但 Set 中却只有一个 5 。若要把已存在的代码或 JSON 结构转换为 Set 来使用，这种特性会让转换更轻松。

> `Set` 构造器实际上可以接收任意可迭代对象作为参数。能使用数组是因为它们默认就是可迭代的， Set 与 Map 也是一样。 `Set` 构造器会使用迭代器来提取参数中的值。（可迭代对象与迭代器详见第八章）

你可以使用 `has()` 方法来测试某个值是否存在于 Set 中，就像这样：

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true
console.log(set.has(6));    // false
```

此处的 Set 不包含 6 这个值，因此 `set.has(6)` 会返回 false 。

移除值

也可以从 Set 中将值移除。使用 `delete()` 方法可移除单个值，而调用 `clear()` 方法可清除 Set 中所有值。以下代码展示了二者的作用：

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true

set.delete(5);

console.log(set.has(5));    // false
console.log(set.size);      // 1

set.clear();

console.log(set.has("5"));  // false
console.log(set.size);      // 0
```

在调用 `delete()` 之后，只有 `5` 被移走；而执行 `clear()` 方法后， `set` 就被清空了。

所有这些方法都提供了一个非常简单的机制来追踪有序的唯一值。不过，在给 Set 添加项目之后，要如何对每个项执行操作呢？此时 `forEcah()` 方法就派上用场了。

Set 上的 forEach() 方法

若你曾处理过数组，或许已熟悉了 `forEach()` 方法。 ES5 给数组添加了 `forEach()` 方法，无须建立 `for` 循环就能方便地处理数组中的每一项。该方法被开发者普遍使用，于是 Set 类型也添加了相同方法，其工作方式也一致。

`forEach()` 方法会被传递一个回调函数，该回调接受三个参数：

1. Set 中下个位置的值；
2. 与第一个参数相同的值；
3. 目标 Set 自身。

Set 版本的 `forEach()` 方法与数组版本有个奇怪差异：前者传给回调函数的前两个参数是相同的。虽然看起来像是错误，但这种行为却有个正当理由。

具有 `forEach()` 方法的其他对象（即数组与 Map ）都会给回调函数传递三个参数，前两个参数都分别是下个位置的值与键（给数组使用的键是数值索引）。

然而 Set 并没有键。 ES6 标准的制定者本可以将 Set 版本 `forEach()` 方法的回调函数设定为只接受两个参数，但这会让它不同于另外两个版本的方法。于是他们找到了一种方式让这些回调函数保持参数一致：将 Set 中的每一项同时认定为键与值。这样 Set 版本 `forEach()` 方法回调函数的前两个参数就始终相同了。

除了参数特点的差异外，在 Set 上使用 `forEach()` 方法与在数组上基本相同。这里有些代码展示了该方法如何工作：

```
let set = new Set([1, 2]);

set.forEach(function(value, key, ownerSet) {
    console.log(key + " " + value);
    console.log(ownerSet === set);
});
```

此代码在 Set 的每一项上进行迭代，并输出了传递给 `forEach()` 的回调函数的值。回调函数每次执行时， `key` 与 `value` 总是相同的，并且 `ownerSet` 也始终等于 `set` 。此代码输出：

```
1 1
true
2 2
true
```

与使用数组相同，如果想在回调函数中使用 `this` ，你可以给 `forEach()` 传入一个 `this` 值作为第二个参数：

```
let set = new Set([1, 2]);

let processor = {
    output(value) {
        console.log(value);
    },
    process(dataSet) {
        dataSet.forEach(function(value) {
            this.output(value);
        }, this);
    }
};

processor.process(set);
```

本例中 `processor.process()` 方法在 Set 上调用了 `forEach()` ，并传递了当前 `this` 作为回调函数的 `this` 值。 这个传递十分必要，这样 `this.output()` 就能正确地解析到 `processor.output()` 方法。此处 `forEach()` 的回调函数仅使用了第一个参数 `value` ，而省略了其余参数。你也可以使用箭头函数来达到相同效果，而不必传入第二个参数，就像这样：

```
let set = new Set([1, 2]);

let processor = {
    output(value) {
        console.log(value);
    },
    process(dataSet) {
        dataSet.forEach((value) => this.output(value));
    }
};

processor.process(set);
```

本例中的箭头函数读取了包含它的 `process()` 函数的 `this` 值，因此就能正确地将 `this.output()` 解析为调用 `processor.output()` 。

要记住，虽然 Set 能非常好地追踪值，并且 `forEach()` 可以让你按顺序处理每一项，但却无法像数组那样用索引来直接访问某个值。若有需要，最好的选择是将 Set 转换为数组。

将 Set 转换为数组

将数组转换为 Set 相当容易，因为可以将数组传递给 `Set` 构造器；而使用扩展运算符也能简单地将 Set 转换回数组。第三章介绍的扩展运算符（ `...` ），能将数组中的项分割开并作为函数的分离参数。你同样能将扩展运算符用于 Set 之类的可迭代对象，将它们转换为数组。例如：

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];

console.log(array);             // [1,2,3,4,5]
```

此处的 Set 在初始化时载入了一个包含重复值的数组。 Set 清除了重复值之后，又使用了扩展运算符将自身的项放到一个新数组中。而这个 Set 仍然包含在创建时所接收的项（ `1` 、 `2` 、 `3` 、 `4` 与 `5` ），这些项只是被复制到了新数组中，而并未从 Set 中消失。

若已存在一个数组，而你想用它创建一个无重复值的新数组时，该方法十分有用。例如：

```
function eliminateDuplicates(items) {
    return [...new Set(items)];
}

let numbers = [1, 2, 3, 3, 3, 4, 5],
    noDuplicates = eliminateDuplicates(numbers);

console.log(noDuplicates);      // [1,2,3,4,5]
```

在 `eliminateDuplicates()` 函数中， Set 只是一个临时的中介物，以便在创建一个无重复的数组之前过滤重复值。

Weak Set

由于 `Set` 类型存储对象引用的方式，它也可以被称为 Strong Set （强引用的 Set ）。对象存储在 `Set` 的一个实例中，实际上相当于存储在变量中。只要对于 `Set` 实例的引用仍然存在，所存储的对象就无法被垃圾回收机制回收，从而无法释放内存。例如：

```
let set = new Set(),
    key = {};

set.add(key);
console.log(set.size);      // 1

// 取消原始引用
key = null;

console.log(set.size);      // 1

// 重新获得原始引用
key = [...set][0];
```

在本例中，将 `key` 设置为 `null` 清除了对 `key` 对象的一个引用，但是另一个引用还存于 `set` 内部。你仍然可以使用扩展运算符将 Set 转换为数组，然后访问数组的第一项，让 `key` 变量取回原先的对象。这种结果在大部分程序中是没问题的，但有时会期望当其他引用消失之后， Set 内部的引用最好也能随之消失。例如，当 JS 代码在网页中运行并保持了与 DOM 元素的联系，在该元素可能被其他脚本移除的情况下，你应当不希望自己的代码保留对该 DOM 元素的最后一个引用（这种情况被称为内存泄漏）。

为了缓解这个问题， ES6 也引入了 **Weak Set** ，该类型只允许存储对象弱引用，而不能存储基本类型的值。对象的弱引用在它自己成为该对象的唯一引用时，不会阻止垃圾回收，

创建 Weak Set

Weak Set 使用 `WeakSet` 构造器来创建，并包含 `add()` 方法、 `has()` 方法以及 `delete()` 方法。以下例子使用了这三个方法：

```
let set = new WeakSet(),
    key = {};

// 将对象加入 set
set.add(key);

console.log(set.has(key));      // true

set.delete(key);

console.log(set.has(key));      // false
```

使用 Weak Set 很像在使用正规的 Set 。你可以在 Weak Set 上添加、移除或检查引用，也可以给构造器传入一个可迭代对象来初始化 Weak Set 的值：

```
let key1 = {},
    key2 = {},
    set = new WeakSet([key1, key2]);

console.log(set.has(key1));     // true
console.log(set.has(key2));     // true
```

在本例中，一个数组被传给了 `WeakSet` 构造器。由于该数组包含了两个对象，这些对象就被添加到了 Weak Set 中。要记住若数组中包含了非对象的值，就会抛出错误，因为 `WeakSet` 构造器不接受基本类型的值。

Set 类型之间的关键差异

Weak Set 与正规 Set 之间最大的区别是对象的弱引用。此处有个例子说明了这种差异：

```
let set = new WeakSet(),
    key = {};

// 将对象加入 set
set.add(key);

console.log(set.has(key));      // true

// 移除对于键的最后一个强引用，同时从 Weak Set 中移除
key = null;
```

当此代码被执行后， Weak Set 中的 `key` 引用就不能再访问了。这一点无法核实，因为需要把对于该对象的一个引用传递给 `has()` 方法（而只要存在其他引用， Weak Set 内部的弱引用就不会消失）。这会让 Weak Set 的引用特征难以测试，但你可以信任 JS 引擎，引用已正确地被移除了。

这些例子演示了 Weak Set 与正规 Set 的一些共有特征，但它们还有一些关键的差异：

1. 对于 `WeakSet` 的实例，若调用 `add()` 方法时传入了非对象的参数，就会抛出错误，`has()` 或 `delete()` 则会在传入了非对象的参数时返回 `false` ；
2. Weak Set 不可迭代，因此不能被用于 `for-of` 循环；
3. Weak Set 无法暴露出任何迭代器（例如 `keys()` 与 `values()` 方法），因此没有任何编程手段可用于判断 Weak Set 的内容；
4. Weak Set 没有 `forEach()` 方法；
5. Weak Set 没有 `size` 属性。

Weak Set 看起来功能很有限，而这对于正确管理内存而言是必要的。一般来说，若只想追踪对象的引用，应当使用 Weak Set 而不是正规 Set 。

Set 给了你处理值列表的新方式，不过它无法给这些值添加额外信息。因此 ES6 也添加了 Map 类型。

ES6 的 Map

ES6 的 `Map` 类型是键值对的有序列表，而键和值都可以是任意类型。键的比较使用的是 `Object.is()` ，`5` 与 `"5"` 的类型不同，因此你能将它们同时作为键来使用。这与使用对象来模拟 Map 截然不同，因为对象的属性会被强制转换为字符串。

你可以调用 `set()` 方法并给它传递一个键与一个关联的值，来给 Map 添加项；此后使用键名来调用 `get()` 方法便能提取对应的值。例如：

```
let map = new Map();
map.set("title", "Understanding ES6");
map.set("year", 2016);

console.log(map.get("title"));      // "Understanding ES6"
console.log(map.get("year"));       // 2016
```

此例存储了两个键值对。 `"title"` 键存储了一个字符串，而 `"year"` 键则存储了一个数值，此后调用 `get()` 方法提取出了二者的值。如果指定键不存在于 Map 中， 则 `get()` 方法就会返回特殊值 `undefined` 。

你也可以将对象作为键，这也是之前使用对象来模拟 Map 的变通方法所无法做到的。此处有个例子：

```
let map = new Map(),
    key1 = {},
    key2 = {};

map.set(key1, 5);
map.set(key2, 42);

console.log(map.get(key1));         // 5
console.log(map.get(key2));         // 42
```

此代码使用了对象 `key1` 与 `key2` 作为 Map 的键，并存储了两个不同的值。由于这些键不会被强制转换成其他形式，每个对象就都被认为是唯一的键。这允许你给对象关联额外数据，而无须修改对象自身。

Map 的方法

Map 与 Set 有意共享了几个方法，允许你使用相似方式来与 Map 及 Set 进行交互。以下三个方法在 Map 与 Set 上都存在：

- `has(key)` ：判断指定的键是否存在于 Map 中；
- `delete(key)` ：移除 Map 中的键以及对应的值；
- `clear()` ：移除 Map 中所有的键与值。

Map 同样拥有 `size` 属性，用于指明包含了多少个键值对。以下代码用不同方式使用了这三种方法以及 `size` 属性：

```
let map = new Map();
map.set("name", "Nicholas");
map.set("age", 25);

console.log(map.size);          // 2

console.log(map.has("name"));   // true
console.log(map.get("name"));   // "Nicholas"

console.log(map.has("age"));    // true
console.log(map.get("age"));    // 25

map.delete("name");
console.log(map.has("name"));   // false
console.log(map.get("name"));   // undefined
console.log(map.size);          // 1

map.clear();
console.log(map.has("name"));   // false
console.log(map.get("name"));   // undefined
console.log(map.has("age"));    // false
console.log(map.get("age"));    // undefined
console.log(map.size);          // 0
```

与用于 Set 时一样， `size` 属性总是包含了 Map 中键值对的数量。此例中的 `Map` 实例起初有 `"name"` 与 `"age"` 两个键，因此传递这两个键给 `has()` 方法都会返回 `true` 。在 `"name"` 键被 `delete()` 方法移除后，使用 `"name"` 调用 `has()` 方法就会返回 `false` ，并且 `size` 属性表明 Map 的项减少了一个。之后 `clear()` 方法移除了残存的键，此时用两个键调用 `has()` 方法都会返回 `false` ，而 `size` 属性则变成了 0 。

`clear()` 方法是从 Map 中移除大量数据的快速方法，同样也有一次性将大量数据添加到 Map 的方法：

Map 的初始化

依然类似于 Set ，你能将数组传递给 `Map` 构造器，以便使用数据来初始化一个 Map 。该数组中的每一项也必须是数组，内部数组的首个项会作为键，第二项则为对应值。这样整个 Map 就被这些双项数组所填充。例如：

```
let map = new Map([["name", "Nicholas"], ["age", 25]]);

console.log(map.has("name"));   // true
console.log(map.get("name"));   // "Nicholas"
console.log(map.has("age"));    // true
console.log(map.get("age"));    // 25
console.log(map.size);          // 2
```

通过构造器中的初始化， `"name"` 与 `"age"` 这两个键就被添加到 `map` 变量中。虽然由数组构成的数组看起来有点奇怪，这对于准确表示键来说却是必要的：因为键允许是任意数据类型，将键存储在数组中，是确保它们在被添加到 Map 之前不会被强制转换为其他类型的唯一方法。

Map 上的 forEach 方法

Map 的 `forEach()` 方法类似于 Set 与数组的同名方法，它接受一个能接收三个参数的回调函数：

1. Map 中下个位置的值；
2. 该值所对应的键；
3. 目标 Map 自身。

回调函数的这些参数更紧密契合了数组 `forEach()` 方法的行为，即：第一个参数是值、第二个参数则是键（数组中的键是数值索引）。此处有个示例：

```
let map = new Map([ ["name", "Nicholas"], ["age", 25]]);

map.forEach(function(value, key, ownerMap) {
    console.log(key + " " + value);
    console.log(ownerMap === map);
});
```

`forEach()` 的回调函数输出了传给它的信息。其中 `value` 与 `key` 被直接输出， `ownerMap` 与 `map` 进行了比较，说明它们是相等的。这样输出了：

```
name Nicholas
true
age 25
true
```

传递给 `forEach()` 的回调函数接收了每个键值对，按照键值对被添加到 Map 中的顺序。这种行为与在数组上调用 `forEach()` 方法有所不同，后者的回调函数会按数值索引的顺序接收到每一个项。

> 你也可以给 `forEach()` 提供第二个参数来指定回调函数中的 `this` 值，其行为与 Set 版本的 `forEach()` 一致。

Weak Map

Weak Map 对 Map 而言，就像 Weak Set 对 Set 一样： Weak 版本都是存储对象弱引用的方式。在 **Weak Map** 中，所有的键都必须是对象（尝试使用非对象的键会抛出错误），而且这些对象都是弱引用，不会干扰垃圾回收。当 Weak Map 中的键在 Weak Map 之外不存在引用时，该键值对会被移除。

Weak Map 的最佳用武之地，就是在浏览器中创建一个关联到特定 DOM 元素的对象。例如，某些用在网页上的 JS 库会维护一个自定义对象，用于引用该库所使用的每一个 DOM 元素，并且其映射关系会存储在内部的对象缓存中。

该方法的困难之处在于：如何判断一个 DOM 元素在网页中已不复存在，以便该库能移除此元素的关联对象。若做不到，该库就会继续保持对 DOM 元素的一个无效引用，并造成内存泄漏。使用 Weak Map 来追踪 DOM 元素，依然允许将自定义对象关联到每个 DOM 元素，而在此对象所关联的 DOM 元素不复存在时，它就会在 Weak Map 中被自动销毁。

> 必须注意的是， Weak Map 的键才是弱引用，而值不是。在 Weak Map 的值中存储对象，就算该对象的其他引用已全都被移除，也会阻止垃圾回收。

使用 **Weak Map**

ES6 的 `WeakMap` 类型是键值对的无序列表，其中键必须是非 null 的对象，值则允许是任意类型。 `WeakMap` 的接口与 `Map` 的非常相似，都使用 `set()` 与 `get()` 方法来分别添加与提取数据：

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

let value = map.get(element);
console.log(value);             // "Original"

// 移除元素
element.parentNode.removeChild(element);
element = null;

// 该 Weak Map 在此处为空
```

此例存储了一个键值对。 `element` 键是一个 DOM 元素，用于存储一个关联的字符串值。将此 DOM 元素传递给 `get()` 方法，就能提取对应的值。随后将此 DOM 元素从页面文档中移除、并且将引用它的变量设置为 `null` ，则对应的数据也就会在 Weak Map 中被移除。

类似于 Weak Set ，没有任何办法可以确认 Weak Map 是否为空，因为它没有 `size` 属性。在其他引用被移除后，由于对键的引用不再有残留，也就无法调用 `get()` 方法来提取对应的值。 Weak Map 已经切断了对于该值的访问，其所占的内存在垃圾回收器运行时便会被释放。

Weak Map 的初始化

为了初始化 Weak Map ，需要把一个由数组构成的数组传递给 `WeakMap` 构造器。就像正规 Map 构造器那样，每个内部数组都应当有两个项，第一项是作为键的非 null 的对象，第二项则是对应的值（允许任意类型）。例如：

```
let key1 = {},
    key2 = {},
    map = new WeakMap([[key1, "Hello"], [key2, 42]]);

console.log(map.has(key1));      // true
console.log(map.get(key1));      // "Hello"
console.log(map.has(key2));      // true
console.log(map.get(key2));      // 42
```

对象 `key1` 与 `key2` 被用作 Weak Map 的键, `get()` 与 `has()` 方法能访问它们。在传递给 `WeakMap` 构造器的参数中，若任意键值对使用了非对象的键，构造器就会抛出错误。

Weak Map 的方法

Weak Map 只有两个附加方法能用来与键值对交互。 `has()` 方法用于判断指定的键是否存在于 Map 中，而 `delete()` 方法则用于移除一个特定的键值对。与 Weak Set 相同，枚举 Weak Map 既没必要也不可能，因此 `clear()` 方法也不存在。以下例子同时用到了 `has()` 与 `delete()` 方法：

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

console.log(map.has(element));   // true
console.log(map.get(element));   // "Original"

map.delete(element);
console.log(map.has(element));   // false
console.log(map.get(element));   // undefined
```

此处一个 DOM 元素再次在 Weak Map 中被作为键来使用。 `has()` 方法能够查看一个引用是否被用作了 Weak Map 的键。但需要注意，必须要有对于该键的另一个非空引用，才能使用此方法。而使用 `delete()` 方法则会把键从 Weak Map 中强制移除，此后用该键调用 `has()`

方法就会返回 `false` ， `get()` 方法则会返回 `undefined` 。

对象的私有数据

虽然大多数开发者认为 Weak Map 的主要用途是关联数据与 DOM 元素，但仍然还存在许多可能的用法（并且无疑仍有一些用法尚未被发现）。 Weak Map 的一个实际应用是在对象实例中存储私有数据。在 ES6 中对象的所有属性都是公开的，因此若想让数据仅对于对象自身可访问，那么就需要一些创造力。研究以下例子：

```
function Person(name) {
    this._name = name;
}

Person.prototype.getName = function() {
    return this._name;
};
```

此代码使用了公共约定的下划线来表示私有属性，以表明一个成员应当被认为是私有的，不应从对象实例外进行修改。此处的意图是只允许用 `getName()` 来访问 `this._name` ，而不允许 `_name` 的值被修改。然而，毫无办法阻止任何人写入 `_name` 属性，所以它依然能够被有意或无意地改写。

在 ES5 中能够创建几乎真正私有的数据，只要在创建对象时使用类似下面的模式：

```
var Person = (function() {

    var privateData = {},
        privateId = 0;

    function Person(name) {
        Object.defineProperty(this, "_id", { value: privateId++ });

        privateData[this._id] = {
            name: name
        };
    }

    Person.prototype.getName = function() {
        return privateData[this._id].name;
    };

    return Person;
}());
```

此例用 IIFE 包裹了 `Person` 的定义，其中含有两个私有变量： `privateData` 与 `privateId` 。 `privateData` 对象存储了所有实例的私有信息，而 `privateId` 则被用于为每个实例产生一个唯一 ID 。 当 `Person` 构造器被调用时，一个不可枚举、不可配置、不可写入的 `_id` 属性

就被添加了。

接下来在 `privateData` 对象中建立了与实例 ID 对应的一个入口，其中存储着 `name` 的值。随后在 `getName()` 函数中，就能使用 `this._id` 作为 `privateData` 的键来提取该值。由于 `privateData` 无法从 IIFE 外部进行访问，实际的数据就是安全的，尽管 `this._id` 在 `privateData` 对象上依然是公开暴露的。

此方式的最大问题在于 `privateData` 中的数据永不会消失，因为在对象实例被销毁时没有任何方法可以获知该数据， `privateData` 对象就将永远包含多余的数据。这个问题现在可以改用 Weak Map 来解决了，如下：

```
let Person = (function() {

    let privateData = new WeakMap();

    function Person(name) {
        privateData.set(this, { name: name });
    }

    Person.prototype.getName = function() {
        return privateData.get(this).name;
    };

    return Person;
}());
```

此版本的 `Person` 范例使用了 Weak Map 而不是对象来保存私有数据。由于 `Person` 对象的实例本身能被作为键来使用，于是也就无须再记录单独的 ID 。当 `Person` 构造器被调用时，将 `this` 作为键在 Weak Map 上建立了一个入口，而包含私有信息的对象成为了对应的值，其中只存放了 `name` 属性。通过将 `this` 传递给 `privateData.get()` 方法，以获取值对象并访问其 `name` 属性， `getName()` 函数便能提取私有信息。这种技术让私有信息能够保持私有状态，并且当与之关联的对象实例被销毁时，私有信息也会被同时销毁。

Weak Map 的用法与局限性

当需要在 Weak Map 与正规 Map 中作出抉择时，首要考虑因素在于你是否只想使用对象类型的键。若是，那么最好的选择就是 Weak Map ，因为它能确保额外数据在不再可用后被销毁，从而能优化内存使用并规避内存泄漏。

要记住 Weak Map 只为它们的内容提供了很低的可见度，因此你不能使用 `forEach()` 方法、 `size` 属性或 `clear()` 方法来管理其中的项。如果你确实需要一些检测功能，那么正规 Map 会是更好的选择，只是一定要确保留意内存的使用。

当然，若你只是想使用非对象的键，那么正规 Map 就是唯一选择。

总结

ES6 正式将 Set 与 Map 引入了 JS 。在此之前，开发者往往使用对象来模拟它们，但由于与对象属性有关的限制，经常会碰壁。

Set 是无重复值的有序列表。根据 `Object.is()` 方法来判断值的不等性，以确保无重复。Set 会自动移除重复的值，因此你可以使用它来过滤数组中的重复值并返回结果。 Set 并不是数组的子类型，所以你无法随机访问其中的值。但你可以使用 `has()` 方法来判断某个值是否存在于 Set 中，或通过 `size` 属性来查看其中有多少个值。 `Set` 类型还拥有 `forEach()` 方法，用于处理每个值。

Weak Set 是只能包含对象的特殊 Set 。其中的对象使用弱引用来存储，意味着当 Weak Set 中的项是某个对象的仅存引用时，它不会阻碍垃圾回收。由于内存管理的复杂性， Weak Set 的内容不能被检查，因此最好仅将 Weak Set 用于追踪需要被归组在一起的对象。

Map 是有序的键值对，其中的键允许是任何类型。与 Set 相似，通过调用 `Object.is()` 方法来判断重复的键，这意味着能将数值 `5` 与字符串 `"5"` 作为两个相对独立的键。使用 `set()` 方法能将任何类型的值关联到某个键上，并且此后能用 `get()` 方法将该值提取出来。 Map 也拥有一个 `size` 属性与一个 `forEach()` 方法，让项目访问更容易。

Weak Map 是一种特殊的 Map ，包含的键只能是对象类型。与 Weak Set 相似，键的对象引用是弱引用，因此当它是某个对象的仅存引用时，也不会阻碍垃圾回收。当键被回收之后，所关联的值也同时从 Weak Map 中被移除。对于与对象相关联的附加信息来说，若要在访问它们的代码之外对其进行生命周期管理，则 Weak Map 在内存管理方面的特性让它们成为了唯一合适的选择。

第八章 迭代器与生成器

许多编程语言都将迭代数据的方式从使用 `for` 循环转变到使用迭代器对象， `for` 循环需要初始化变量以便追踪集合内的位置，而迭代器则以编程方式返回集合中的下一个项。迭代器能使操作集合变得更简单，因此 ES6 也将其添加到 JS 中。当新的数组方法与新的集合类型（例如 Set 与 Map ）结合时，迭代器就是高效数据处理的关键。并且在 JS 语言的很多新成分中也能找到迭代器：新增的 `for-of` 与它协同工作，扩展运算符（ `...` ）也使用了它，而它甚至还能让异步操作更易完成。

本章涵盖了迭代器的许多用法，但首先来说，重点要理解迭代器被加入 JS 的背后历史。

循环的问题

如果你曾用 JS 编写过程序，那么或许写过如下代码：

```
var colors = ["red", "green", "blue"];

for (var i = 0, len = colors.length; i < len; i++) {
    console.log(colors[i]);
}
```

此处使用了 `for` 循环的标准方式，借助 `i` 变量来追踪 `colors` 数组中的位置索引。当 `i` 的值小于暂存在 `len` 变量中的数组长度时，循环每一次执行都会递增 `i` 的值。

虽然这个循环非常直观，然而若它被嵌套使用而要追踪多个变量时，复杂度就会提高，由此也更容易引发错误。相似的 `for` 循环代码会被写在多个地方，一不小心就会写错循环变量。迭代器正是用来解决此问题的。

何为迭代器？

迭代器是被专用设计用于迭代的对象，带有特定接口。所有的迭代器对象都拥有 `next()` 方法，调用时会返回一个结果对象。结果对象有两个属性：对应下一个值的 `value` ，以及一个布尔类型的 `done` ，其值为 `true` 时表示没有更多值可供使用。迭代器持有一个指向集合位置的内部指针，每当调用了 `next()` 方法，迭代器就会返回相应的下一个值。

若在最后一个值被返回后再调用 `next()` ，所返回的 `done` 属性值会是 `true` ，并且 `value` 属性值会是迭代器自身的返回值（ **return value** ，即使用 return 语句明确返回的值）。该返回值不是原数据集的一部分，却会成为迭代器数据的最后一个片段。它类似于函数的返回值，是向调用者返回信息的最后手段，在未提供时会使用 `undefined` 。

记住这些特性后，在 ES5 中创建一个迭代器就相当简单了：

```
function createIterator(items) {

    var i = 0;

    return {
        next: function() {

            var done = (i >= items.length);
            var value = !done ? items[i++] : undefined;

            return {
                done: done,
                value: value
            };

        }
    };
}

var iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// 之后的所有调用
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

`createIterator()` 函数返回一个带有 `next()` 方法的对象。每当调用此方法时， `items` 数组的下一个值就会作为 `value` 属性的值被返回。当 `i` 的值为 3 时， `done` 属性变成 `true` ，并且利用三元运算符将 `value` 指定为 `undefined` 。 `done` 与 `value` 二者最终的结果符合了 ES6 迭代器规则最后的特殊情况，也就是在数据所有片段都被使用之后的迭代器上调用 `next()` 方法。

正如此例所示，根据 ES6 制定的规则来书写迭代器稍微有点复杂。

幸好， ES6 还提供了生成器，让创建迭代器对象变得更简单。

何为生成器？

生成器（ **generator** ）是能返回一个迭代器的函数。生成器函数由放在 `function` 关键字之后的一个星号（ `*` ）来表示，并能使用新的 `yield` 关键字。将星号紧跟在 `function` 关键字之后，或是在中间留出空格，这两种写法都没问题，正如下例：

```
// 生成器
function *createIterator() {
    yield 1;
    yield 2;
    yield 3;
}

// 生成器能像正规函数那样被调用，但会返回一个迭代器
let iterator = createIterator();

console.log(iterator.next().value);     // 1
console.log(iterator.next().value);     // 2
console.log(iterator.next().value);     // 3
```

`createIterator()` 前面的星号让此函数变成一个生成器。 `yield` 关键字也是 ES6 新特性，指定了 `next()` 方法调用迭代器时应当按顺序返回的值。此例所生成的迭代器能够在 `next()` 方法调用成功时返回三个不同的值：依次是 `1` 、 `2` 、 `3` 。生成器能像其他任意函数那样被调用，正如示例中创建 `iterator` 的代码。

生成器函数最有意思的特性可能就是它们会在每个 `yield` 语句后停止执行。例如，此代码中 `yield 1` 执行后，该函数将不会再执行任何操作，直到迭代器的 `next()` 方法被调用才继续执行 `yield 2` 。在函数中停止执行的能力分外强大，并能引出生成器函数的一些有趣的用法（详见"迭代器高级功能"一节）。

`yield` 关键字可用于值或表达式，因此你可以通过生成器给迭代器添加项目，而不是机械式地将项目一个个列出。作为一个例子，此处给出了在 `for` 循环内使用 `yield` 的方法：

```
function *createIterator(items) {
    for (let i = 0; i < items.length; i++) {
        yield items[i];
    }
}

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// 之后的所有调用
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此例传递了一个名为 `items` 的数组给 `createIterator()` 生成器函数。在此函数内， `for` 循环在循环执行时从数组中返回元素给迭代器。每当遇到 `yield` ，循环就会停止；而每当 `iterator` 上的 `next()` 方法被调用，循环就会继续执行下一条 `yield` 语句。

生成器函数是 ES6 的一个重要特性，它能被用于所有可使用函数的位置。本节剩余部分会聚焦在书写生成器的其他有用方法上。

> `yield` 关键字只能用在生成器内部，用于其他任意位置都是语法错误，即使在生成器内部嵌套的函数中也不行，正如此例：
>
> ```
> function *createIterator(items) {
>
>   items.forEach(function(item) {
>
>       // 语法错误
>       yield item + 1;
>   });
> }
> ```
>
> 尽管 `yield` 严格位于 `createIterator()` 内部，但它无法穿越函数边界，因此这段代码仍然有语法错误。在内嵌函数中 `yield` 与 `return` 非常相似，二者均不能用作外层函数的返回语句。

生成器函数表达式

你可以使用函数表达式来创建一个生成器，只要在 `function` 关键字与圆括号之间使用一个星号（ `*` ）即可。例如：

```
let createIterator = function *(items) {
    for (let i = 0; i < items.length; i++) {
        yield items[i];
    }
};

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// 之后的所有调用
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此代码中的 `createIterator()` 是一个生成器函数表达式，而不是一个函数声明。这个函数表达式是匿名的，因此星号只能放置在 `function` 关键字与圆括号之间。除此之外，此例与前一个版本的 `createIterator()` 函数没有区别，都使用了一个 `for` 循环。

> 不能用箭头函数创建生成器。

生成器对象方法

由于生成器就是函数，因此也可以被添加到对象中。例如，你可以在 ES5 风格的对象字面量中使用函数表达式来创建一个生成器：

```
var o = {

    createIterator: function *(items) {
        for (let i = 0; i < items.length; i++) {
            yield items[i];
        }
    }
};

let iterator = o.createIterator([1, 2, 3]);
```

你也可以使用 ES6 方法的速记法，只要在方法名之前加上一个星号（ `*` ）：

```
var o = {

    *createIterator(items) {
        for (let i = 0; i < items.length; i++) {
            yield items[i];
        }
    }
};

let iterator = o.createIterator([1, 2, 3]);
```

这些例子的功能等价于“生成器函数表达式”小节中的例子，只是语法有区别。在速记法版本中，由于 `createIterator()` 方法没有使用 `function` 关键字来定义，星号紧贴在方法名之前书写，不过其实在星号与方法名之间可以留出空格。

可迭代对象与 for-of 循环

可迭代对象（ **iterable** ）是包含 `Symbol.iterator` 属性的对象，与迭代器紧密相关。这个 `Symbol.iterator` 知名符号定义了为指定对象返回迭代器的函数。在 ES6 中，所有的集合对象（数组、 Set 与 Map ）以及字符串都是可迭代对象，因此它们都有默认的迭代器。可迭代对象被设计用于与 ES 新增的 `for-of` 循环配合使用。

> 生成器默认会为 `Symbol.iterator` 属性赋值，因此它创建的所有迭代器都是可迭代对象。

在本章开头我曾提到过在 `for` 循环中追踪索引的问题。迭代器是解决此问题的第一部分； `for-of` 循环则是第二部分：它完全消除了追踪集合索引的必要，让你无拘束地专注于操作集合内容。

`for-of` 循环在循环每次执行时会调用可迭代对象的 `next()` 方法，并将结果对象的 `value` 值存储在一个变量上。循环过程会持续到结果对象的 `done` 属性变成 `true` 为止。此处有个范例：

```
let values = [1, 2, 3];

for (let num of values) {
    console.log(num);
}
```

此代码输出了如下内容：

```
1
2
3
```

这里的 `for-of` 循环首先调用了 `values` 数组的 `Symbol.iterator` 方法，获取了一个迭代器，这个调用由 JS 引擎在后台进行。接下来 `iterator.next()` 被调用，迭代器结果对象的 `value` 属性被读出并放入了 `num` 变量，依次为 1 、 2 、 3 。当结果对象的 `done` 变成 `true` ，循环立刻结束，因此 `num` 绝不会被赋值为 `undefined` 。

若仅需迭代数组或集合的值，那么使用 `for-of` 循环而非 `for` 循环就是个好主意。 `for-of` 循环一般不易出错，因为需要留意的条件更少；传统的 `for` 循环则被用于处理更复杂的控制条件。

> 在不可迭代对象、 `null` 或 `undefined` 上使用 `for-of` 语句，会抛出错误。

访问默认迭代器

你可以使用 `Symbol.iterator` 来访问对象上的默认迭代器，就像这样：

```
let values = [1, 2, 3];
let iterator = values[Symbol.iterator]();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此代码获取了 `values` 数组的默认迭代器，并用它来迭代数组中的项。这与使用 `for-of` 循环时在后台发生的过程一致。

既然 `Symbol.iterator` 指定了默认迭代器，你就可以用其来检测一个对象是否能被迭代，正如下例：

```
function isIterable(object) {
    return typeof object[Symbol.iterator] === "function";
}

console.log(isIterable([1, 2, 3]));     // true
console.log(isIterable("Hello"));       // true
console.log(isIterable(new Map()));     // true
console.log(isIterable(new Set()));     // true
console.log(isIterable(new WeakMap())); // false
console.log(isIterable(new WeakSet())); // false
```

这个 `isIterable()` 函数仅仅查看对象是否存在一个类型为函数的默认迭代器。 `for-of` 循环在执行之前会做类似的检查。

本节至今的范例已经展示了如何在内置的可迭代类型上使用 `Symbol.iterator` ，但你也能用 `Symbol.iterator` 属性来自定义可迭代对象。

创建可迭代对象

开发者自定义对象默认情况下不是可迭代对象，但你可以创建一个包含生成器的 `Symbol.iterator` 属性，将它们变成可迭代对象。例如：

```
let collection = {
    items: [],
    *[Symbol.iterator]() {
        for (let item of this.items) {
            yield item;
        }
    }

};

collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}
```

此代码输出了如下内容：

```
1
2
3
```

本例首先用 `Symbol.iterator` 方法（注意其名称之前也有星号）创建了生成器，为 `collection` 对象定义了一个默认的迭代器。接下来该生成器使用了一个 `for-of` 循环来对 `this.items` 中的值进行迭代，并使用了 `yield` 来返回每个值。 `collection` 对象依靠 `this.items` 的默认迭代器来工作，而不是对预定义的值进行手动迭代以产生返回结果。

> 本章后面的“生成器委托”会描述一种不同方法，能使用别的对象的迭代器。

现在你已经看到了数组默认迭代器的一些用法，但 ES6 中还内置了更多的迭代器，以便更轻易地处理数据集合。

内置的迭代器

迭代器是 ES6 的一个重要部分，正因如此，语言已默认为许多内置类型创建了迭代器，你无须自行实现。只有当内置的迭代器无法满足需求时，才有必要创建自定义迭代器，这通常发生在定义你自己的对象或类时。最常用的迭代器或许就存在于集合上。

集合的迭代器

ES6 具有三种集合对象类型：数组、 Map 与 Set 。它们都拥有如下的迭代器，便于探索其内容：

- `entries()` ：返回一个包含键值对的迭代器；
- `values()` ：返回一个包含集合中的值的迭代器；
- `keys()` ：返回一个包含集合中的键的迭代器。

你可以调用上述方法之一来提取集合中的迭代器。

entries() 迭代器

`entries()` 迭代器会在每次 `next()` 被调用时返回一个双项数组，表示集合中每个元素的键与值：对于数组来说，第一项是数值索引；而对于 Set ，第一项既是键又是值； Map 的第一项也是键。

此处有使用此迭代器的一些范例：

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ES6");
data.set("format", "ebook");

for (let entry of colors.entries()) {
    console.log(entry);
}

for (let entry of tracking.entries()) {
    console.log(entry);
}

for (let entry of data.entries()) {
    console.log(entry);
}
```

调用 `console.log()` 输出了以下内容：

```
[0, "red"]
[1, "green"]
[2, "blue"]
[1234, 1234]
[5678, 5678]
[9012, 9012]
["title", "Understanding ES6"]
["format", "ebook"]
```

此代码在每种集合类型上都使用了 `entries()` 方法来提取迭代器，并且使用 `for-of` 循环来迭代它们的项。此处控制台的输出说明了每个对象的键与值是如何被成对返回的。

values() 迭代器

`values()` 迭代器仅仅能返回存储在集合内的值，例如：

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ES6");
data.set("format", "ebook");

for (let value of colors.values()) {
    console.log(value);
}

for (let value of tracking.values()) {
    console.log(value);
}

for (let value of data.values()) {
    console.log(value);
}
```

此代码输出了如下内容：

```
"red"
"green"
"blue"
1234
5678
9012
"Understanding ES6"
"ebook"
```

正如本例所显示的，调用 `values()` 迭代器返回了每种类型中包含的准确数据，而没有这些数据在集合内的任意位置信息。

keys() 迭代器

`keys()` 迭代器能返回集合中的每一个键。对于数组来说，它只返回了数值类型的键，永不返回数组的其他自有属性； Set 的键与值是相同的，因此它的 `keys()` 与 `values()` 会返回相同的迭代器；对于 Map ， `keys()` 迭代器返回了每个不重复的键。此处有个例子演示了这三种情况：

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ES6");
data.set("format", "ebook");

for (let key of colors.keys()) {
    console.log(key);
}

for (let key of tracking.keys()) {
    console.log(key);
}

for (let key of data.keys()) {
    console.log(key);
}
```

本例输出了如下内容：

```
0
1
2
1234
5678
9012
"title"
"format"
```

`keys()` 迭代器获取了 `colors` 、 `tracking` 与 `data` 各自的键，这些键在三个 `for-of` 循环中被打印出来。对于数组对象来说，只有数值类型索引被打印了，即使你向数组添加了具名属性也依然如此。这与在数组上使用 `for-in` 循环是不同的，因为 `for-in` 循环会迭代所有属性而不仅是数值索引。

集合类型的默认迭代器

当 `for-of` 循环没有显式指定迭代器时，每种集合类型都有一个默认的迭代器供循环使用。 `values()` 方法是数组与 Set 的默认迭代器，而 `entries()` 方法则是 Map 的默认迭代器。在 `for-of` 循环中使用集合对象时，这些默认迭代器会让处理更容易一些。作为例子，研究如下代码：

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ES6");
data.set("format", "print");

// 与使用 colors.values() 相同
for (let value of colors) {
    console.log(value);
}

// 与使用 tracking.values() 相同
for (let num of tracking) {
    console.log(num);
}

// 与使用 data.entries() 相同
for (let entry of data) {
    console.log(entry);
}
```

此处没有明确指定迭代器，因此会使用默认的迭代器函数。数组、 Set 与 Map 的默认迭代器反映了这些对象是如何被初始化的，于是此代码会输出如下内容：

```
"red"
"green"
"blue"
1234
5678
9012
["title", "Understanding ES6"]
["format", "print"]
```

默认情况下，数组与 Set 输出了它们的值，而 Map 返回的则是可以直接传给 `Map` 构造器的数组格式。另一方面， Weak Set 与 Weak Map 并未拥有内置的迭代器，使用弱引用意味着无法获知这些集合内部到底有多少个值，同时意味着没有方法可以迭代这些值。

> 解构与 `for-of` 循环
>
> Map 默认迭代器的行为有助于在 `for-of` 循环中使用解构，正如此例：
>
> ```
> let data = new Map();
>
> data.set("title", "Understanding ES6");
> data.set("format", "ebook");
>
> // 与使用 data.entries() 相同
> for (let [key, value] of data) {
>   console.log(key + "=" + value);
> }
> ```
>
> 此代码中的 `for-of` 循环使用了数组解构，来将 Map 中的每个项存入 `key` 与 `value` 变量。使用这种方式，你能轻易地同时处理键与值，而无须访问一个双项数组，或是回到 Map 中去获取键或值。在 Map 上进行 `for-of` 循环时使用数组解构，能让循环就像处理 Set 或数组那样方便。

字符串的迭代器

从 ES5 发布开始， JS 的字符串就慢慢变得越来越像数组。例如 ES5 标准化了字符串的方括号表示法，用于访问其中的字符（即：使用 `text[0]` 来获取第一个字符，以此类推） 。不过方括号表示法工作在码元而非字符上，因此它不能被用于正确访问双码元的字符，正如此例所演示的：

```
var message = "A  B" ;

for (let i=0; i < message.length; i++) {
    console.log(message[i]);
}
```

此代码使用了方括号表示法与 `length` 属性来迭代字符串并打印字符，该字符串包含一个 Unicode 字符，输出结果有点出人意料：

```
A
(blank)
(blank)
(blank)
(blank)
B
```

> 译注： `(blank)` 代表空行。

由于四字节字符被当作两个分离的码元来对待，此处的输出在 `A` 与 `B` 之间就有了四个空行。

幸好，ES6 旨在为 Unicode 提供完全支持（详见第二章），字符串的默认迭代器正是用于解决字符串迭代问题，借助它就能处理字符而不是码元。将上个范例修改为使用字符串默认迭代器配合 `for-of` 循环，会得到更加合适的输出。以下是调整之后的代码：

```
var message = "A  B" ;

for (let c of message) {
    console.log(c);
}
```

此代码输出了如下内容：

```
A
(blank)

(blank)
B
```

作用对象是字符，让本次的结果更符合预期，循环成功地打印出了这个 Unicode 字符以及其余字符。

NodeList 的迭代器

文档对象模型（DOM）具有一种 `NodeList` 类型，用于表示页面文档中元素的集合。对于需要书写在浏览器中运行的 JS 代码的开发者，要理解 `NodeList` 对象与数组之间的差异总是稍有困难。 `NodeList` 对象与数组都使用了 `length` 属性来标明项的数量，并且都使用方括号表示法来访问各个项。然而本质上来说， `NodeList` 与数组的行为是非常不同的，这会引发许多混乱。

随着默认迭代器被加入 ES6，DOM 关于 `NodeList` 的规定也包含了一个默认迭代器（此规定在 HTML 规范而非 ES6 规范中），其表现方式与数组的默认迭代器一致。这意味着你可以将 `NodeList` 用于 `for-of` 循环，或用于其他使用对象默认迭代器的场合。例如：

```
var divs = document.getElementsByTagName("div");

for (let div of divs) {
    console.log(div.id);
}
```

此代码调用 `getElementsByTagName()` 来获取一个包含 `document` 对象中的所有 `<div>` 元素的 `NodeList` 。接下来 `for-of` 循环迭代了每个元素并打印出它们的 ID ，实际上这段代码与在标准数组上使用时并无二致。

扩展运算符与非数组的可迭代对象

回顾一下第七章，可以用扩展运算符（ `...` ）将一个 Set 转换为数组，例如：

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];

console.log(array);             // [1,2,3,4,5]
```

此代码在数组字面量中使用扩展运算符，以便将 `set` 中的值填充到数组。扩展运算符能作用于所有可迭代对象，并且会使用默认迭代器来判断需要使用哪些值。所有的值都从迭代器中被读取出来，并按返回值的顺序插入数组。此例工作正常是由于 Set 是可迭代对象，而这种方式同样还能用于任意的可迭代对象。此处有另一个例子：

```
let map = new Map([ ["name", "Nicholas"], ["age", 25]]),
    array = [...map];

console.log(array);         // [ ["name", "Nicholas"], ["age", 25]]
```

此处的扩展运算符将 `map` 转换为一个由数组构成的数组。由于 Map 的默认迭代器返回的是键值对，最终的数组看起来与调用 `new Map()` 时所传入的参数一模一样。

你能不限次数地在数组字面量中使用扩展运算符，而且可以在任意位置用扩展运算符将可迭代对象的多个项插入数组，按扩展运算符所在的位置插入新数组，例如：

```
let smallNumbers = [1, 2, 3],
    bigNumbers = [100, 101, 102],
    allNumbers = [0, ...smallNumbers, ...bigNumbers];

console.log(allNumbers.length);     // 7
console.log(allNumbers);    // [0, 1, 2, 3, 100, 101, 102]
```

此处的扩展运算符使用 `smallNumbers` 与 `bigNumbers` 中的数据来创建 `allNumbers` 数组。在 `allNumbers` 被创建时，值在其中的排列顺序与数组被添加的顺序一致： 首先是 `0` ，其次是来自 `smallNumbers` 数组的元素，最后是来自 `bigNumbers` 数组的元素。原始数组并没有被改变，只是它们的值被复制到了 `allNumbers` 数组中。

既然扩展运算符能用在任意可迭代对象上，它就成为了将可迭代对象转换为数组的最简单方法。你可以将字符串转换为包含字符（而非码元）的数组，也能将浏览器中的 `NodeList` 对象转换为节点数组。

现在你已基本了解迭代器是如何工作的，包括使用 `for-of` 或扩展运算符的场合，是时候去看看迭代器的一些更复杂用法了。

迭代器高级功能

使用迭代器的基本功能，并使用生成器来方便地创建迭代器，你就已经可以完成很多工作了。不过，除了单纯迭代集合的值，在其他任务中迭代器的作用更强大。在 ES6 的开发过程中，出现了许多独特的思想与模式，激励着规范制定者去添加更多的功能。这些附加功能可能很细微，但将它们结合使用就能形成一些有趣的交互。

传递参数给迭代器

本章中的范例已经展示了迭代器能够将值传递出来，通过 `next()` 方法或者在生成器中使用 `yield` 都可以。但你还能通过 `next()` 方法向迭代器内传递参数。当一个参数被传递给 `next()` 方法时，该参数就会成为生成器内部 `yield` 语句的值。这种能力对于诸如异步编程之类的高级功能来说是非常重要的。此处有个基本范例：

```
function *createIterator() {
    let first = yield 1;
    let second = yield first + 2;       // 4 + 2
    yield second + 3;                   // 5 + 3
}

let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next(4));          // "{ value: 6, done: false }"
console.log(iterator.next(5));          // "{ value: 8, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

对于 `next()` 的首次调用是一个特殊情况，传给它的任意参数都会被忽略。由于传递给 `next()` 的参数会成为 `yield` 语句的值，则首次调用给 `next()` 提供的参数就只会替换生成器函数中的第一个 `yield` 语句；但若要在生成器内部使用该值，又要求它在此 `yield` 语句之前就必须能被访问到，而这是不可能的。因此在首次调用 `next()` 时传递参数是没有意义的。

> 译注：译者不赞成原文的说法，首次调用 `next()` 不能传参的原因应当是——
>
> 由于传递给 `next()` 的参数会成为 `yield` 语句的值，该 `yield` 语句指的是上次生成器中断执行处的语句；而 `next()` 方法第一次被调用时，生成器函数才刚刚开始执行，没有所谓的“上一次中断处的 `yield` 语句”可供赋值。

在第二次调用 `next()` 时， `4` 作为参数被传递进去，这个 `4` 最终被赋值给了生成器函数内部的 `first` 变量。在包含赋值操作的第一个 `yield` 语句中，表达式右侧在第一次调用 `next()` 时被计算，而表达式左侧则在第二次调用 `next()` 方法时、并在生成器函数继续执行前被计算。由于第二次调用 `next()` 传入了 `4` ，这个值就被赋给了 `first` 变量，之后生成器继续执行。

第二个 `yield` 使用了第一个 `yield` 的结果并加上了 2 ，也就是返回了一个 `6` 。当 `next()` 被第三次调用时，传入了参数 `5` 。这个值被赋给了 `second` 变量，并随后用在了第三个 `yield` 语句中，返回了 `8` 。

通过考虑在生成器函数内部每次继续运行时都执行了什么代码，会有助于思考到底发生了什么。示意图 8-1 用颜色演示了代码在 yield 之前是如何执行的。

```
next()           function *createIterator() {
next(4)              let first = yield 1;
next(5)              let second = yield first + 2;
                     yield second + 3;
                 }
```

示意图 8-1: 在一个生成器内部的代码执行

黄色表示对于 `next()` 的第一次调用、以及在生成器内部执行的代码；水蓝色表示了对 `next(4)` 的调用以及随之执行的代码；而紫色则表示对 `next(5)` 的调用以及随之执行的代码。其中难点在于：在左侧代码执行之前，右侧的各个表达式是如何执行与停止的。这使得调试错综复杂的生成器要比调试正规函数更麻烦一些。

到目前的范例已表明：当一个值传递给 `next()` 方法时， `yield` 的表现相似于 `return` 。然而，在生成器内部不仅能使用这样的执行技巧，还能让迭代器抛出一个错误。

在迭代器中抛出错误

能传递给迭代器的不仅是数据，还可以是错误条件。迭代器可以使用一个 `throw()` 方法，让迭代器在恢复执行时抛出一个错误。这是对异步编程来说很重要的一个能力，同时也会增加生成器内部的灵活度，既能模仿返回一个值，又能模仿抛出错误（这也就是退出函数的两种方式）。你可以传递一个错误对象给 `throw()` 方法，当迭代器继续进行处理时应当抛出此错误。例如：

```
function *createIterator() {
    let first = yield 1;
    let second = yield first + 2;       // yield 4 + 2 ，然后抛出错误
    yield second + 3;                   // 永不会被执行
}

let iterator = createIterator();

console.log(iterator.next());                   // "{ value: 1, done: false }"
console.log(iterator.next(4));                  // "{ value: 6, done: false }"
console.log(iterator.throw(new Error("Boom"))); // 从生成器中抛出了错误
```

在本例中，前两个 `yield` 表达式被照常运算，但当 `throw()` 被调用时，一个错误在 `let second` 运算之前就被抛出了。这有效停止了代码执行，类似于直接抛出错误，其中唯一区别是错误在何处被抛出。示意图 8-2 演示了每一步所执行的是什么代码。

next()
next(4)
throw(new Error());

function *createIterator() {
let first = yield 1;
let second = yield first + 2;
yield second + 3;
}

示意图 8-2: 在一个生成器内部抛出错误

在此示意图中，红色表示当 `throw()` 被调用时所执行的代码，红星说明了错误在生成器内部大约何时被抛出。前两个 `yield` 语句被执行之后，当调用 `throw()` 时，在任何其他代码执行之前错误就被抛出了。

了解这些之后，你就可以在生成器内部使用一个 `try-catch` 块来捕捉这种错误：

```
function *createIterator() {
    let first = yield 1;
    let second;

    try {
        second = yield first + 2;       // yield 4 + 2 ，然后抛出错误
    } catch (ex) {
        second = 6;                     // 当出错时，给变量另外赋值
    }
    yield second + 3;
}

let iterator = createIterator();

console.log(iterator.next());                   // "{ value: 1, done: false }"
console.log(iterator.next(4));                  // "{ value: 6, done: false }"
console.log(iterator.throw(new Error("Boom"))); // "{ value: 9, done: false }"
console.log(iterator.next());                   // "{ value: undefined, done: true }"
```

本例使用一个 `try-catch` 块包裹了第二个 `yield` 语句。尽管这个 `yield` 自身的执行不会出错，但在对 `second` 变量赋值之前，错误就在此时被抛出，于是 `catch` 部分捕捉错误并将这个变量赋值为 6 ，然后再继续执行到下一个 `yield` 处并返回了 9 。

要注意一件有趣的事情发生了： `throw()` 方法就像 `next()` 方法一样返回了一个结果对象。由于错误在生成器内部被捕捉，代码继续执行到下一个 `yield` 处并返回了下一个值，也就是 `9` 。

将 `next()` 与 `throw()` 都当作迭代器的指令，会有助于思考。 `next()` 方法指示迭代器继续执行（可能会带着给定的值），而 `throw()` 方法则指示迭代器通过抛出一个错误继续执行。在调用点之后会发生什么，根据生成器内部的代码来决定。

`next()` 与 `throw()` 方法控制着迭代器在使用 `yield` 时内部的执行，但你也可以使用 `return` 语句。不过 `return` 的工作方式与它在正规函数中有一些差异，正如你将在下一节中看到的那样。

生成器的 Return 语句

由于生成器是函数，你可以在它内部使用 `return` 语句，既可以让生成器早一点退出执行，也可以指定在 `next()` 方法最后一次调用时的返回值。在本章大多数例子中，对迭代器上的 `next()` 的最后一次调用都返回了 `undefined` ，但你还可以像在其他函数中那样，使用 `return` 来指定另一个返回值。在生成器内， `return` 表明所有的处理已完成，因此 `done` 属性会被设为 `true` ，而如果提供了返回值，就会被用于 `value` 字段。此处有个例子，单纯使用 `return` 让生成器更早返回：

```
function *createIterator() {
    yield 1;
    return;
    yield 2;
    yield 3;
}

let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此代码中的生成器在一个 `yield` 语句后跟随了一个 `return` 语句。这个 `return` 表明将不会再有更多值可用，之后的 `yield` 语句也因此不会再执行（它们是不可到达的）。

你也可以指定一个返回值，会被用于最终返回的结果对象中的 `value` 字段。例如：

```
function *createIterator() {
    yield 1;
    return 42;
}

let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 42, done: true }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此处，当第二次调用 `next()` 方法时，值 `42` 被返回在 `value` 字段中，此时 `done` 字段的值才第一次变为了 `true` 。第三次调用 `next()` 返回了一个对象，其 `value` 属性再次变回 `undefined` ，你在 `return` 语句中指定的任意值都只会在结果对象中出现一次，此后 `value` 字段就会被重置为 `undefined` 。

> 扩展运算符与 `for-of` 循环会忽略 `return` 语句所指定的任意值。一旦它们看到 `done` 的值为 `true` ，它们就会停止操作而不会读取对应的 `value` 值。不过，在生成器进行委托时，迭代器的返回值会非常有用。

生成器委托

在某些情况下，将两个迭代器的值合并在一起会更有用。生成器可以用星号（ `*` ）配合 `yield` 这一特殊形式来委托其他的迭代器。正如生成器的定义，星号出现在何处是不重要的，只要落在 `yield` 关键字与生成器函数名之间即可。此处有个范例：

```
function *createNumberIterator() {
    yield 1;
    yield 2;
}

function *createColorIterator() {
    yield "red";
    yield "green";
}

function *createCombinedIterator() {
    yield *createNumberIterator();
    yield *createColorIterator();
    yield true;
}

var iterator = createCombinedIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: "red", done: false }"
console.log(iterator.next());           // "{ value: "green", done: false }"
console.log(iterator.next());           // "{ value: true, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此例中的 `createCombinedIterator()` 生成器依次委托了 `createNumberIterator()` 与 `createColorIterator()` 。返回的迭代器从外部看来就是一个单一的迭代器，用于产生所有的值。每次对 `next()` 的调用都会委托给合适的生成器，直到使用 `createNumberIterator()` 与 `createColorIterator()` 创建的迭代器全部清空为止。然后最后一条 `yield` 语句会被执行以返回 `true` 。

生成器委托也能让你进一步使用生成器的返回值。这是访问这些返回值的最简单方式，并且在执行复杂任务时会非常有用。例如：

```
function *createNumberIterator() {
    yield 1;
    yield 2;
    return 3;
}

function *createRepeatingIterator(count) {
    for (let i=0; i < count; i++) {
        yield "repeat";
    }
}

function *createCombinedIterator() {
    let result = yield *createNumberIterator();
    yield *createRepeatingIterator(result);
}

var iterator = createCombinedIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

此处 `createCombinedIterator()` 生成器委托了 `createNumberIterator()` 并将它的返回值赋值给了 `result` 变量。由于 `createNumberIterator()` 包含 `return 3` 语句，该返回值就是 `3` 。 `result` 变量接下来会作为参数传递给 `createRepeatingIterator()` 生成器，指示同一个字符串需要被重复几次（在本例中是三次）。

注意值 `3` 从未在对于 `next()` 方法的任何调用中被输出。当前它仅仅存在于 `createCombinedIterator()` 生成器内部。但你也可以通过添加另一个 `yield` 语句来输出这个值，正如：

```
function *createNumberIterator() {
    yield 1;
    yield 2;
    return 3;
}

function *createRepeatingIterator(count) {
    for (let i=0; i < count; i++) {
        yield "repeat";
    }
}

function *createCombinedIterator() {
    let result = yield *createNumberIterator();
    yield result;
    yield *createRepeatingIterator(result);
}

var iterator = createCombinedIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: "repeat", done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

在此代码中，额外的 `yield` 语句明确地将 `createNumberIterator()` 生成器的返回值进行了输出。

使用返回值的生成器委托是一种非常强大的范式，能引出一些非常有趣的应用场景，尤其是在与异步操作结合时。

> 你可以直接在字符串上使用 `yield *` （例如 `yield * "hello"` ），字符串的默认迭代器会被使用。

异步任务运行

围绕着生成器的许多兴奋点都与异步编程直接相关。 JS 中的异步编程是一把双刃剑：简单任务很容易用异步实现，但复杂任务就会给代码组织带来痛苦。由于生成器能在执行过程中有效地暂停代码操作，它就给异步编程带来了许多可能性。

执行异步操作的传统方式是调用一个包含回调的函数。例如，考虑在 Node.js 中从磁盘读取一个文件：

```
let fs = require("fs");

fs.readFile("config.json", function(err, contents) {
    if (err) {
        throw err;
    }

    doSomethingWith(contents);
    console.log("Done");
});
```

能使用文件名与一个回调函数去调用 `fs.readFile()` 方法，在读取操作结束之后，回调函数就会被调用。此回调函数查看是否存在错误，无错误时处理返回的 `contents` 数据。当你需要完成的任务很少时，这么做很有效；然而当你需要嵌套回调函数，或者要按顺序处理一系列的异步任务时，此方式就会非常麻烦。在这种场合下，生成器与 `yield` 会带来帮助。

一个简单的任务运行器

由于 `yield` 能停止运行，并在重新开始运行前等待 `next()` 方法被调用，你就可以在没有回调函数的情况下实现异步调用。首先，你需要一个能够调用生成器并启动迭代器的函数，就像这样：

```
function run(taskDef) {

    // 创建迭代器，让它在别处可用
    let task = taskDef();

    // 启动任务
    let result = task.next();

    // 递归使用函数来保持对 next() 的调用
    function step() {

        // 如果还有更多要做的
        if (!result.done) {
            result = task.next();
            step();
        }
    }

    // 开始处理过程
    step();

}
```

`run()` 函数接受一个任务定义（即一个生成器函数）作为参数，它会调用生成器来创建一个迭代器，并将迭代器存放在 `task` 变量上。 `task` 变量放在函数的外层，因此它可以被函数内的其他函数访问到。第一次对 `next()` 的调用启动了迭代器，并将结果存储下来以便稍后使用。 `step()` 函数查看 `result.done` 的值，如果为 false 就在递归调用自身之前调用 `next()` 方法。每次调用 `next()` 都会把返回的结果保存在 `result` 变量上，它总是会被最新的信息所重写。对于 `step()` 的初始调用启动了处理过程，该过程会查看 `result.done` 来判断是否还有更多的工作要做。

配合这个已实现的 `run()` 函数，你就可以运行一个包含多条 `yield` 语句的生成器，就像这样：

```
run(function*() {
    console.log(1);
    yield;
    console.log(2);
    yield;
    console.log(3);
});
```

此例只是将三个数值输出到控制台，单纯用于表明对 `next()` 的所有调用都已被执行。然而，仅仅使用几次 yield 并不太有意义，下一步是要把值传进迭代器并获取返回数据。

带数据的任务运行

传递数据给任务运行器最简单的方式，就是把 `yield` 返回的值传入下一次的 `next()` 调用。为此，你仅需传递 `result.value` ，正如以下代码：

```
function run(taskDef) {

    // 创建迭代器，让它在别处可用
    let task = taskDef();

    // 启动任务
    let result = task.next();

    // 递归使用函数来保持对 next() 的调用
    function step() {

        // 如果还有更多要做的
        if (!result.done) {
            result = task.next(result.value);
            step();
        }
    }

    // 开始处理过程
    step();

}
```

现在 `result.value` 作为参数被传递给了 `next()` ，这样就能在 `yield` 调用之间传递数据了，就像这样：

```
run(function*() {
    let value = yield 1;
    console.log(value);         // 1

    value = yield value + 3;
    console.log(value);         // 4
});
```

此例向控制台输出了两个值： 1 与 4 。数值 1 由 `yield 1` 语句而来，随之 1 又被传回给了 `value` 变量。数值 4 是在 `value` 上加了 3 的结果，并将运算结果再度赋值给 `value` 。现在数据在对 `yield` 的调用之间流动了起来，仅需再来个微小改动，就能支持异步调用。

异步任务运行器

上个例子只是在 `yield` 之间来回传递静态数据，但等待一个异步处理与此稍微有点差异。任务运行器需要了解回调函数，并了解如何使用它们。并且由于 `yield` 表达式将它们的值传递给了任务运行器，这就意味着任意函数调用都必须返回一个值，并以某种方式标明该返回值是个异步操作调用，而任务运行器应当等待此操作。

此处是将返回值标明为异步操作的一种方法：

```
function fetchData() {
    return function(callback) {
        callback(null, "Hi!");
    };
}
```

此例的目的是：任何打算让任务运行器调用的函数，都应当返回一个能够执行回调函数的函数。 `fetchData()` 函数所返回的函数能接受一个回调函数作为其参数，当返回的函数被调用时，它会执行回调函数并附加一点额外数据（即 `"Hi!"` 字符串）。该回调函数需要由任务运行器提供，以确保回调函数能与当前的迭代器正确交互。虽然 `fetchData()` 函数是同步的，但你能延迟对回调函数的调用，从而轻易地将它改造为异步函数，就像这样：

```
function fetchData() {
    return function(callback) {
        setTimeout(function() {
            callback(null, "Hi!");
        }, 50);
    };
}
```

此版本的 `fetchData()` 在调用回调函数之前引入了 50 毫秒的延迟，说明此模式在同步或异步代码上都能同样良好运作。你只要保证每个需要被 `yield` 调用的函数都遵循此模式。

在深入理解函数如何标注自己是一个异步处理后，你就可以结合这种模式来改造任务运行器。只要 `result.value` 是一个函数，任务运行器就应当执行它，而不是仅仅将它传递给 `next()` 方法。此处有更新后的代码：

```
function run(taskDef) {

    // 创建迭代器，让它在别处可用
    let task = taskDef();

    // 启动任务
    let result = task.next();

    // 递归使用函数来保持对 next() 的调用
    function step() {

        // 如果还有更多要做的
        if (!result.done) {
            if (typeof result.value === "function") {
                result.value(function(err, data) {
                    if (err) {
                        result = task.throw(err);
                        return;
                    }

                    result = task.next(data);
                    step();
                });
            } else {
                result = task.next(result.value);
                step();
            }

        }
    }

    // 开始处理过程
    step();

}
```

使用 `===` 运算符判断出 `result.value` 是个函数时，将一个回调函数作为参数去调用它。该回调函数遵循了 Node.js 的惯例，将任何潜在错误作为第一个参数（ `err` ）传入，而处理结果则作为第二个参数。若 `err` 非空，也就表示有错误发生，需要使用该错误对象去调用 `task.throw()` ，而不是调用 `task.next()` ，这样错误就会在恰当的位置被抛出；若不存在错误， `data` 参数将会被传入 `task.next()` ，而其调用结果也会被保存下来。接下来，调用 `step()` 来继续处理过程。若 `result.value` 并非函数，它就会被直接传递给 `next()` 方法。

这个新版任务运行器已经为所有的异步任务准备好了。为了在 Node.js 中从一个文件读取数据，你需要创建对于 `fs.readFile()` 的一个封装，它类似于本节起始处的 `fetchData()` 函数，能返回另一个函数。例如：

```
let fs = require("fs");

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}
```

这个 `readFile()` 方法接受单个参数，即文件名，并返回一个能执行回调函数的函数。此回调函数会被直接传递给 `fs.readFile()` 方法，后者会在操作完成后执行回调。接下来你就可以使用 `yield` 来运行这个任务，如下：

```
run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

此例执行了异步的 `readFile()` 操作，而在主要代码中并未暴露出任何回调函数。除了 `yield` 之外，此代码看起来与同步代码并无二致。既然执行异步操作的函数都遵循了同一接口，你就可以用貌似同步的代码来书写处理逻辑。

当然，这些范例中所使用的模式也有缺点：你无法完全确认一个能返回函数的函数是异步的。但现在唯一重要的是让你理解任务运行背后的原理。 promise 提供了更强有力的方式来调度异步任务，第十一章会进一步介绍这个主题。

总结

迭代器是 ES6 重要的一部分，并且也是语言若干关键元素的依赖。表面上看，迭代器只是提供了一种使用简单接口来返回值序列的简单方式。然而，在 ES6 中使用迭代器，还有着种种更加复杂的方式。

`Symbol.iterator` 符号被用于定义对象的默认迭代器。内置对象与开发者自定义对象都可以使用这个符号，以提供一个能返回迭代器的方法。当一个对象具有 `Symbol.iterator` 时，它就会被认为是可迭代对象。

`for-of` 循环在循环中使用可迭代对象来返回一系列数据。与使用传统 `for` 循环进行迭代相比，使用 `for-of` 要容易得多，因为你不再需要追踪计数器并控制循环何时结束。 `for-of` 循环会自动从迭代器中读取所有数据，直到没有更多数据为止，然后退出循环。

为了让 `for-of` 更易使用， ES6 中的许多类型都具有默认的迭代器。所有的集合类型（也就是数组、 Map 与 Set ）都具有迭代器，让它们的内容更易被访问。字符串同样具有一个默认迭代器，能更加轻易地迭代字符串中的字符（而非码元）。

扩展运算符能操作任意的可迭代对象，同时也能从一个迭代器中依次读取数据，更简单地将其转换为数组。

生成器是一个特殊的函数，可以在被调用时自动创建一个迭代器。生成器的定义用一个星号（ `*` ）来表示，使用 `yield` 关键字能指明在每次成功的 `next()` 方法调用时应当返回什么值。

生成器委托促进了对迭代器行为的良好封装，让你能将已有的生成器重用在新的生成器中。通过调用 `yield *` 而非 `yield` ，你就能把已有生成器用在其他生成器内部。这种处理方式能将多个迭代器的返回值合并到新迭代器上。

生成器与迭代器最有趣、最令人激动的方面，或许就是能创建外观清晰的异步操作代码。你不必到处使用回调函数，而是可以建立貌似同步的代码，但实际上却使用 `yield` 来等待异步操作结束。

第九章 JS的类

与大多数正规的面向对象编程语言不同， JS 从创建之初就不支持类，也没有把类继承作为定义相似对象以及关联对象的主要方式，这让不少开发者感到困惑。而从 ES1 诞生之前直到 ES5 时期，很多库都创建了一些工具，使 JS 看起来仿佛能支持类。尽管一些 JS 开发者强烈认为这门语言不需要类，但为处理类而创建的代码库如此之多，导致 ES6 最终引入了类。

在探索 ES6 的类的过程中，理解类的潜在机制会很有帮助，因此本章将会首先讨论 ES5 的开发者如何实现对类行为的模仿。然而正如你将在后面看到的， ES6 的类并不与其他语言的类完全相同，所具备的独特性正配合了 JS 的动态本质。

ES5 中的仿类结构

JS 在 ES5 及更早版本中都不存在类。与类最接近的是：创建一个构造器，然后将方法指派到该构造器的原型上。这种方式通常被称为创建一个自定义类型。例如：

```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
};

let person = new PersonType("Nicholas");
person.sayName();   // 输出 "Nicholas"

console.log(person instanceof PersonType);  // true
console.log(person instanceof Object);      // true
```

此代码中的 `PersonType` 是一个构造器函数，并创建了单个属性 `name` 。 `sayName()` 方法被指派到原型上，因此在 `PersonType` 对象的所有实例上都共享了此方法。接下来，使用 `new` 运算符创建了 `PersonType` 的一个新实例 `person` ，此对象能被判定是通过原型而继承了 `PersonType` 与 `Object` 的实例。

这种基本模式在许多对类进行模拟的 JS 库中都存在，而这也是 ES6 类的出发点。

类的声明

类在 ES6 中最简单的形式就是类声明，它看起来很像其他语言中的类。

基本的类声明

类声明以 `class` 关键字开始，其后是类的名称；剩余部分的语法看起来就像对象字面量中的方法简写，并且在方法之间不需要使用逗号。作为范例，此处有个简单的类声明：

```
class PersonClass {

    // 等价于 PersonType 构造器
    constructor(name) {
        this.name = name;
    }

    // 等价于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName();   // 输出 "Nicholas"

console.log(person instanceof PersonClass);     // true
console.log(person instanceof Object);          // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName);  // "function"
```

这个 `PersonClass` 类声明的行为非常类似上个例子中的 `PersonType` 。类声明允许你在其中使用特殊的 `constructor` 方法名称直接定义一个构造器，而不需要先定义一个函数再把它当作构造器使用。由于类的方法使用了简写语法，于是就不再需要使用 `function` 关键字。`constructor` 之外的方法名称则没有特别的含义，因此可以随你高兴自由添加方法。

> 自有属性（ **Own properties** ）：该属性出现在实例上而不是原型上，只能在类的构造器或方法内部进行创建。在本例中， `name` 就是一个自有属性。我建议应在构造器函数内创建所有可能出现的自有属性，这样在类中声明变量就会被限制在单一位置（有助于代码检查）。

有趣的是，相对于已有的自定义类型声明方式来说，类声明仅仅是以它为基础的一个语法糖。 `PersonClass` 声明实际上创建了一个拥有 `constructor` 方法及其行为的函数，这也是 `typeof PersonClass` 会得到 `"function"` 结果的原因。此例中的 `sayName()` 方法最终也成为 `PersonClass.prototype` 上的一个方法，类似于上个例子中 `sayName()` 与 `PersonType.prototype` 之间的关系。这些相似处允许你把自定义类型与类混合使用，而不必被具体的用法困扰。

为何要使用类的语法

尽管类与自定义类型之间有相似性，但仍然要记住一些重要的区别：

1. 类声明不会被提升，这与函数定义不同。类声明的行为与 `let` 相似，因此在程序执行到声明处之前，类都会位于暂时性死区内。
2. 类声明中的所有代码会自动运行并锁定在严格模式下。

3. 类的所有方法都是不可枚举的，这是对于自定义类型的显著变化，后者必须用 `Object.defineProperty()` 才能将方法改变为不可枚举。
4. 类的所有方法内部都没有 `[[Construct]]` ，因此使用 `new` 来调用它们会抛出错误。
5. 调用类构造器时不使用 `new` ，会抛出错误。
6. 试图在类的方法内部重写类名，会抛出错误。

这样看来，上例中的 `PersonClass` 声明实际上就直接等价于以下未使用类语法的代码：

```
// 直接等价于 PersonClass
let PersonType2 = (function() {

    "use strict";

    const PersonType2 = function(name) {

        // 确认函数被调用时使用了 new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonType2.prototype, "sayName", {
        value: function() {

            // 确认函数被调用时没有使用 new
            if (typeof new.target !== "undefined") {
                throw new Error("Method cannot be called with new.");
            }

            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonType2;
}());
```

首先要注意这里有两个 `PersonType2` 声明：一个在外部作用域的 `let` 声明，一个在 IIFE 内部的 `const` 声明。这说明了为何类名不能在类的方法内被重写，而允许在外部重写。构造器函数检查了 `new.target` ，以保证是使用 `new` 进行调用的，否则就抛出错误。接下来， `sayName()` 方法被定义为不可枚举，并且此方法也检查了 `new.target` ，它则要保证在被调用时没有使用 `new` 。最后一步是将构造器函数返回出去。

此例说明了不使用新语法也能实现类的任何特性，不过类语法显著简化了所有功能的代码。

> 不变的类名
>
> 只有在类的内部，类名才被视为是使用 `const` 声明的。这意味着你可以在外部重写类名，但不能在类的方法内部这么做。例如：
>
> ```
> class Foo {
>   constructor() {
>       Foo = "bar";    // 执行时抛出错误
>   }
> }
>
> // 但在类声明之后没问题
> Foo = "baz";
> ```
>
> 在此代码中，类构造器内部的 `Foo` 与在类外部的 `Foo` 是不同的绑定。内部的 `Foo` 就像是用 `const` 定义的，不能被重写，当构造器尝试使用任何值重写 `Foo` 时，都会抛出错误。但由于外部的 `Foo` 就像是用 `let` 声明的，你可以随时重写类名。

类表达式

类与函数相似之处在于都有两种形式：声明与表达式。函数声明与类声明都以适当的关键词为起始（分别是 `function` 与 `class` ），随后是标识符（即函数名或类名）。函数具有一种表达式形式，无须在 `function` 后面使用标识符；类似的，类也有不需要标识符的表达式形式。类表达式被设计用于变量声明，或可作为参数传递给函数。

基本的类表达式

此处是与上例中的 `PersonClass` 等效的类表达式，随后的代码使用了它：

```
let PersonClass = class {

    // 等价于 PersonType 构造器
    constructor(name) {
        this.name = name;
    }

    // 等价于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};

let person = new PersonClass("Nicholas");
person.sayName();   // 输出 "Nicholas"

console.log(person instanceof PersonClass);     // true
console.log(person instanceof Object);          // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName); // "function"
```

正如此例所示，类表达式不需要在 `class` 关键字后使用标识符。除了语法差异，类表达式的功能等价于类声明。

> 使用类声明还是类表达式，主要是代码风格问题。相对于函数声明与函数表达式之间的区别，类声明与类表达式都不会被提升，因此对代码运行时的行为影响甚微。

具名类表达式

上一节的示例使用了一个匿名的类表达式，不过就像函数表达式那样，你也可以为类表达式命名。为此需要在 `class` 关键字后添加标识符，就像这样：

```
let PersonClass = class PersonClass2 {

    // 等价于 PersonType 构造器
    constructor(name) {
        this.name = name;
    }

    // 等价于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};

console.log(typeof PersonClass);        // "function"
console.log(typeof PersonClass2);       // "undefined"
```

此例中的类表达式被命名为 `PersonClass2` 。 `PersonClass2` 标识符只在类定义内部存在，因此只能用在类方法内部（例如本例的 `sayName()` 内）。在类的外部， `typeof PersonClass2` 的结果为 `"undefined"` ，这是因为外部不存在 `PersonClass2` 绑定。要理解为何如此，请查看未使用类语法的等价声明：

```
// 直接等价于 PersonClass 具名的类表达式
let PersonClass = (function() {

    "use strict";

    const PersonClass2 = function(name) {

        // 确认函数被调用时使用了 new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonClass2.prototype, "sayName", {
        value: function() {

            // 确认函数被调用时没有使用 new
            if (typeof new.target !== "undefined") {
                throw new Error("Method cannot be called with new.");
            }

            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonClass2;
}());
```

创建具名的类表达式， JS 引擎的内部实现稍微有了变化。对于类声明来说，用 `let` 定义的外部绑定与用 `const` 定义的内部绑定有着相同的名称。而类表达式可在内部使用 `const` 来定义它的不同名称，于是此处的 `PersonClass2` 就只能在类的内部使用。

尽管具名类表达式的行为异于具名函数表达式，但它们之间仍然有许多相似点。二者都能被当作值来使用，存在多种利用可能，接下来我将会对此进行介绍。

作为一等公民的类

在编程中，能被当作值来使用的就称为一等公民（ **first-class citizen** ），意味着它能作为参数传给函数、能作为函数返回值、能用来给变量赋值。 JS 的函数就是一等公民（它们有时又被称为一等函数），这是 JS 的独特之处。

ES6 延续了传统，让类同样成为一等公民。这就使得类可以被多种方式所使用。例如，它能作为参数传入函数：

```
function createObject(classDef) {
    return new classDef();
}

let obj = createObject(class {

    sayHi() {
        console.log("Hi!");
    }
});

obj.sayHi();        // "Hi!"
```

此例中的 `createObject()` 函数被调用时接收了一个匿名类表达式作为参数，使用 `new` 创建了该类的一个实例，并将其返回回出来。随后变量 `obj` 储存了所返回的实例。

类表达式的另一个有趣用途是立即调用类构造器，用于创建单例（ Singleton ）。为此，你必须使用 `new` 来配合类表达式，并在表达式后面添加括号。例如：

```
let person = new class {

    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }

}("Nicholas");

person.sayName();       // "Nicholas"
```

此处创建了一个匿名类表达式，并立即执行了它。此模式允许你使用类语法来创建单例，从而不留下任何可被探查的类引用（回忆一下 `PersonClass` 的例子，匿名类表达式只在类的内部创建了绑定，而外部无绑定）。类表达式后面的圆括号表示要调用前面的函数，并且还允许传入参数。

本章至今的例子都集中于带有方法的类，但你还能在类上创建访问器属性，所用的语法类似于对象字面量。

访问器属性

自有属性需要在类构造器中创建，而类还允许你在原型上定义访问器属性。为了创建一个 getter ，要使用 `get` 关键字，并要与后方标识符之间留出空格；创建 setter 用相同方式，只是要改用 `set` 关键字。例如：

```
class CustomHTMLElement {

    constructor(element) {
        this.element = element;
    }

    get html() {
        return this.element.innerHTML;
    }

    set html(value) {
        this.element.innerHTML = value;
    }
}

var descriptor = Object.getOwnPropertyDescriptor(CustomHTMLElement.prototype, "html");
console.log("get" in descriptor);   // true
console.log("set" in descriptor);   // true
console.log(descriptor.enumerable); // false
```

此代码中的 `CustomHTMLElement` 类用于包装一个已存在的 DOM 元素。它的属性 `html` 拥有 getter 与 setter ，委托了元素自身的 `innerHTML` 方法。该访问器属性被创建在 `CustomHTMLElement.prototype` 上，并且像其他类属性那样被创建为不可枚举属性。非类的等价表示如下：

```
// 直接等价于上个范例
let CustomHTMLElement = (function() {

    "use strict";

    const CustomHTMLElement = function(element) {

        // 确认函数被调用时使用了 new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.element = element;
    }

    Object.defineProperty(CustomHTMLElement.prototype, "html", {
        enumerable: false,
        configurable: true,
        get: function() {
            return this.element.innerHTML;
        },
        set: function(value) {
            this.element.innerHTML = value;
        }
    });

    return CustomHTMLElement;
}());
```

正如之前的例子，此例说明了使用类语法能够少写大量的代码。仅仅为 `html` 访问器属性定义的代码量，就几乎相当于等价的类声明的全部代码量了。

可计算的成员名

对象字面量与类之间的相似点还不仅前面那些。类方法与类访问器属性也都能使用可计算的名称。语法与对象字面量相同，不是使用标识符，而是用方括号来包裹一个表达式。例如：

```
let methodName = "sayName";

class PersonClass {

    constructor(name) {
        this.name = name;
    }

    [methodName]() {
        console.log(this.name);
    }
}

let me = new PersonClass("Nicholas");
me.sayName();           // "Nicholas"
```

此版本的 `PersonClass` 使用了一个变量来命名类定义内的方法。字符串 `"sayName"` 被赋值给了 `methodName` 变量，而 `methodName` 变量则被用于声明此后能被直接访问的 `sayName()` 方法。

访问器属性能以相同方式使用可计算的名称，就像这样：

```
let propertyName = "html";

class CustomHTMLElement {

    constructor(element) {
        this.element = element;
    }

    get [propertyName]() {
        return this.element.innerHTML;
    }

    set [propertyName](value) {
        this.element.innerHTML = value;
    }
}
```

此处 `html` 的 getter 与 setter 被设置为需使用 `propertyName` 变量，访问此属性的使用依然是 `.html` ，此处影响的只有定义方式。

你已经看到了在类与对象字面量之间有许多相似点，包括方法、访问器属性、可计算的名称。此外还有一个相似点需要介绍，即生成器。

生成器方法

第八章介绍了生成器，你已学会如何在对象字面量上定义一个生成器：只要在方法名称前附加一个星号（ `*` ）。这一语法对类同样有效，允许将任何方法变为一个生成器。此处有个范例：

```
class MyClass {

    *createIterator() {
        yield 1;
        yield 2;
        yield 3;
    }

}

let instance = new MyClass();
let iterator = instance.createIterator();
```

此代码创建了一个拥有 `createIterator()` 生成器的 `MyClass` 类。该方法返回了一个迭代器，它的值在生成器内部用硬编码提供。当你使用一个对象来表示值的集合、并要求能简单迭代这些值，那么生成器方法就非常有用。数组、 Set 与 Map 都拥有多个生成器方法，负责让开发者用多种方式来操作它们的项。

既然生成器方法很有用，那么在表示集合的自定义类中定义一个默认迭代器，那就更好。你可以使用 `Symbol.iterator` 来定义生成器方法，从而定义出类的默认迭代器，就像这样：

```
class Collection {

    constructor() {
        this.items = [];
    }

    *[Symbol.iterator]() {
        yield *this.items.values();
    }
}

var collection = new Collection();
collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}

// 输出：
// 1
// 2
// 3
```

此例为生成器方法使用了一个可计算名称，并将此方法委托到 `this.items` 数组的 `values()` 迭代器上。任意管理集合的类都包含一个默认迭代器，这是因为一些集合专用的操作都要求目标集合具有迭代器。现在， `Collection` 的任意实例都可以在 `for-of` 循环内被直接使用，也能配合扩展运算符使用。

当你想让方法与访问器属性在对象实例上出现时，就应当把它们添加到类的原型上。而另一方面，若想将方法与访问器属性绑定到类自身，那么你就需要使用静态成员。

静态成员

直接在构造器上添加额外方法来模拟静态成员，这在 ES5 及更早版本中是另一个通用的模式。例如：

```
function PersonType(name) {
    this.name = name;
}

// 静态方法
PersonType.create = function(name) {
    return new PersonType(name);
};

// 实例方法
PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = PersonType.create("Nicholas");
```

在其他编程语言中，工厂方法 `PersonType.create()` 会被认定为一个静态方法，它的数据不依赖 `PersonType` 的任何实例。ES6 的类简化了静态成员的创建，只要在方法与访问器属性的名称前添加正式的 `static` 标注。作为一个例子，此处有个与上例等价的类：

```
class PersonClass {

    // 等价于 PersonType 构造器
    constructor(name) {
        this.name = name;
    }

    // 等价于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }

    // 等价于 PersonType.create
    static create(name) {
        return new PersonClass(name);
    }
}

let person = PersonClass.create("Nicholas");
```

`PersonClass` 的定义拥有名为 `create()` 的单个静态方法，此语法与 `sayName()` 基本相同，只多了一个 `static` 关键字。你能在类中的任何方法与访问器属性上使用 `static` 关键字，唯一限制是不能将它用于 `constructor` 方法的定义。

> 静态成员不能用实例来访问，你始终需要直接用类自身来访问它们。

使用派生类进行继承

ES6 之前，实现自定义类型的继承是个繁琐的过程。严格的继承要求有多个步骤。例如，研究以下范例：

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function Square(length) {
    Rectangle.call(this, length, length);
}

Square.prototype = Object.create(Rectangle.prototype, {
    constructor: {
        value:Square,
        enumerable: true,
        writable: true,
        configurable: true
    }
});

var square = new Square(3);

console.log(square.getArea());              // 9
console.log(square instanceof Square);      // true
console.log(square instanceof Rectangle);   // true
```

`Square` 继承了 `Rectangle` ，为此它必须使用 `Rectangle.prototype` 所创建的一个新对象来重写 `Square.prototype` ，并且还要调用 `Rectangle.call()` 方法。这些步骤常常会搞晕 JS 的新手，并会成为有经验开发者出错的根源之一。

类让继承工作变得更轻易，使用熟悉的 `extends` 关键字来指定当前类所需要继承的函数，即可。生成的类的原型会被自动调整，而你还能调用 `super()` 方法来访问基类的构造器。此处是与上个例子等价的 ES6 代码：

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {

        // 与 Rectangle.call(this, length, length) 相同
        super(length, length);
    }
}

var square = new Square(3);

console.log(square.getArea());              // 9
console.log(square instanceof Square);      // true
console.log(square instanceof Rectangle);   // true
```

此次 `Square` 类使用了 `extends` 关键字继承了 `Rectangle` 。 `Square` 构造器使用了 `super()` 配合指定参数调用了 `Rectangle` 的构造器。注意与 ES5 版本的代码不同， `Rectangle` 标识符仅在类定义时被使用了（在 `extends` 之后）。

继承了其他类的类被称为派生类（ **derived classes** ）。如果派生类指定了构造器，就需要使用 `super()` ，否则会造成错误。若你选择不使用构造器， `super()` 方法会被自动调用，并会使用创建新实例时提供的所有参数。例如，下列两个类是完全相同的：

```
class Square extends Rectangle {
    // 没有构造器
}

// 等价于：

class Square extends Rectangle {
    constructor(...args) {
        super(...args);
    }
}
```

此例中的第二个类展示了与所有派生类默认构造器等价的写法，所有的参数都按顺序传递给了基类的构造器。在当前需求下，这种做法并不完全准确，因为 `Square` 构造器只需要单个参数，因此最好手动定义构造器。

> 使用 `super()` 时需牢记以下几点：
>
> 1. 你只能在派生类中使用 `super()` 。若尝试在非派生的类（即：没有使用 `extends` 关键字的类）或函数中使用它，就会抛出错误。
> 2. 在构造器中，你必须在访问 `this` 之前调用 `super()` 。由于 `super()` 负责初始化 `this` ，因此试图先访问 `this` 自然就会造成错误。
> 3. 若在类的构造器中不调用 `super()` ，唯一避免出错的办法是在构造器中返回一个对象。

屏蔽类方法

派生类中的方法总是会屏蔽基类的同名方法。例如，你可以将 `getArea()` 方法添加到 `Square` 类，以便重定义它的功能：

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // 重写并屏蔽 Rectangle.prototype.getArea()
    getArea() {
        return this.length * this.length;
    }
}
```

由于 `getArea()` 已经被定义为 `Square` 的一部分， `Rectangle.prototype.getArea()` 方法就不能在 `Square` 的任何实例上被调用。当然，你总是可以使用 `super.getArea()` 方法来调用基类中的该方法，就像这样：

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // 重写、屏蔽并调用了 Rectangle.prototype.getArea()
    getArea() {
        return super.getArea();
    }
}
```

用这种方式使用 `super` ，其效果等同于第四章讨论过的 super 引用（详见“使用 super 引用的简单原型访问”）。 `this` 值会被自动设置为正确的值，因此你就能进行简单的调用。

继承静态成员

如果基类包含静态成员，那么这些静态成员在派生类中也是可用的。继承的工作方式类似于其他语言，但对于 JS 而言则是新概念。此处有个范例：

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }

    static create(length, width) {
        return new Rectangle(length, width);
    }
}

class Square extends Rectangle {
    constructor(length) {

        // 与 Rectangle.call(this, length, length) 相同
        super(length, length);
    }
}

var rect = Square.create(3, 4);

console.log(rect instanceof Rectangle);     // true
console.log(rect.getArea());                // 12
console.log(rect instanceof Square);        // false
```

在此代码中，一个新的静态方法 `create()` 被添加到 `Rectangle` 类中。通过继承，该方法会以 `Square.create()` 的形式存在，并且其行为方式与 `Rectangle.create()` 一样。

从表达式中派生类

在 ES6 中派生类的最强大能力，或许就是能够从表达式中派生类。只要一个表达式能够返回一个具有 `[[Construct]]` 属性以及原型的函数，你就可以对其使用 `extends` 。例如：

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());              // 9
console.log(x instanceof Rectangle);    // true
```

`Rectangle` 被定义为 ES5 风格的构造器，而 `Square` 则是一个类。由于 `Rectangle` 具有 `[[Construct]]` 以及原型， `Square` 类就能直接继承它。

`extends` 后面能接受任意类型的表达式，这带来了巨大可能性，例如动态地决定所要继承的类：

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function getBase() {
    return Rectangle;
}

class Square extends getBase() {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());              // 9
console.log(x instanceof Rectangle);    // true
```

`getBase()` 函数作为类声明的一部分被直接调用，它返回了 `Rectangle` ，使得此例的功能等价于前一个例子。并且由于可以动态地决定基类，那也就能创建不同的继承方式。例如，你可以有效地创建混入：

```
let SerializableMixin = {
    serialize() {
        return JSON.stringify(this);
    }
};

let AreaMixin = {
    getArea() {
        return this.length * this.width;
    }
};

function mixin(...mixins) {
    var base = function() {};
    Object.assign(base.prototype, ...mixins);
    return base;
}

class Square extends mixin(AreaMixin, SerializableMixin) {
    constructor(length) {
        super();
        this.length = length;
        this.width = length;
    }
}

var x = new Square(3);
console.log(x.getArea());               // 9
console.log(x.serialize());             // "{"length":3,"width":3}"
```

此例使用了混入（ mixin ）而不是传统继承。 `mixin()` 函数接受代表混入对象的任意数量的参数，它创建了一个名为 `base` 的函数，并将每个混入对象的属性都赋值到新函数的原型上。此函数随后被返回，于是 `Square` 就能够对其使用 `extends` 关键字了。注意由于仍然使用了 `extends` ，你就必须在构造器内调用 `super()` 。

`Square` 的实例既有来自 `AreaMixin` 的 `getArea()` 方法，又有来自 `SerializableMixin` 的 `serialize()` 方法，这是通过原型继承实现的。 `mixin()` 函数使用了混入对象的所有自有属性，动态地填充了新函数的原型（记住：若多个混入对象拥有相同的属性，则只有最后添加的属性会被保留）。

> 任意表达式都能在 `extends` 关键字后使用，但并非所有表达式的结果都是一个有效的类。下列表达式类型就会明确导致错误：
>
> - `null` ；
> - 生成器函数（详见第八章）。
>
> 试图使用结果为上述值的表达式来创建一个新的类实例，都会抛出错误，因为不存在 `[[Construct]]` 可供调用。

继承内置对象

几乎从 JS 数组出现那天开始，开发者就想通过继承机制来创建他们自己的特殊数组类型。在 ES5 及早期版本中，这是不可能做到的。试图使用传统继承并不能产生功能正确的代码，例如：

```
// 内置数组的行为
var colors = [];
colors[0] = "red";
console.log(colors.length);         // 1

colors.length = 0;
console.log(colors[0]);             // undefined

// 在 ES5 中尝试继承数组

function MyArray() {
    Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
    constructor: {
        value: MyArray,
        writable: true,
        configurable: true,
        enumerable: true
    }
});

var colors = new MyArray();
colors[0] = "red";
console.log(colors.length);         // 0

colors.length = 0;
console.log(colors[0]);             // "red"
```

`console.log()` 在此代码尾部的输出说明，对数组使用传统形式的 JS 继承，产生了预期外的行为。 `MyArray` 实例上的 `length` 属性以及数值属性，其行为与内置数组并不一致，因为这些功能并未被涵盖在 `Array.apply()` 或数组原型中。

在 ES6 中的类，其设计目的之一就是允许从内置对象上进行继承。为了达成这个目的，类的继承模型与 ES5 或更早版本的传统继承模型有轻微差异：

在 ES5 的传统继承中， `this` 的值会先被派生类（例如 `MyArray` ）创建，随后基类构造器（例如 `Array.apply()` 方法）才被调用。这意味着 `this` 一开始就是 `MyArray` 的实例，之后才使用了 `Array` 的附加属性对其进行了装饰。

在 ES6 基于类的继承中， `this` 的值会先被基类（ `Array` ）创建，随后才被派生类的构造器（ `MyArray` ）所修改。结果是 `this` 初始就拥有作为基类的内置对象的所有功能，并能正确接收与之关联的所有功能。

以下范例实际展示了基于类的特殊数组：

```
class MyArray extends Array {
    // 空代码块
}

var colors = new MyArray();
colors[0] = "red";
console.log(colors.length);         // 1

colors.length = 0;
console.log(colors[0]);             // undefined
```

`MyArray` 直接继承了 `Array` ，因此工作方式与正规数组一致。与数值索引属性的互动会更新 `length` 属性，而操纵 `length` 属性也能更新索引属性。这意味着你既能适当地继承 `Array` 来创建你自己的派生数组类，也同样能继承其他的内置对象。伴随着这些附加功能，ES6 与派生类型有效解决了从内置类型进行派生这最后的特殊情况，不过这种情况仍然值得继续研究。

Symbol.species 属性

继承内置对象会带来一个有趣特性，任意能返回内置对象实例的方法，在派生类上却会自动返回派生类的实例。因此，若你拥有一个继承了 `Array` 的派生类 `MyArray` ，诸如 `slice()` 之类的方法都会返回 `MyArray` 的实例。例如：

```
class MyArray extends Array {
    // 空代码块
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);      // true
console.log(subitems instanceof MyArray);   // true
```

在此代码中， `slice()` 方法返回了 `MyArray` 的一个实例。 `slice()` 方法是从 `Array` 上继承的，原本应当返回 `Array` 的一个实例。而 `Symbol.species` 属性在后台造成了这种变化。

`Symbol.species` 知名符号被用于定义一个能返回函数的静态访问器属性。每当类实例除了构造器之外的方法必须创建一个实例时，前面返回的函数就被用为新实例的构造器。下列内置类型都定义了 `Symbol.species` ：

- `Array`
- `ArrayBuffer` （详见第十章）
- `Map`
- `Promise`
- `RegExp`
- `Set`
- 类型化数组（详见第十章）

以上每个类型都拥有默认的 `Symbol.species` 属性，其返回值为 `this` ，意味着该属性总是会返回自身的构造器函数。若你准备在一个自定义类上实现此功能，代码就像这样：

```
// 几个内置类型使用 species 的方式类似于此
class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
    }

    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}
```

在此例中， `Symbol.species` 知名符号被用于定义 `MyClass` 的一个静态访问器属性。注意此处只有 getter 而没有 setter ，这是因为修改类的种类是不允许的。任何对 `this.constructor[Symbol.species]` 的调用都会返回 `MyClass` ， `clone()` 方法使用了该定义来返回一个新的实例，而没有直接使用 `MyClass` ，这就允许派生类重写这个值。例如：

```
class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
    }

    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}

class MyDerivedClass1 extends MyClass {
    // 空代码块
}

class MyDerivedClass2 extends MyClass {
    static get [Symbol.species]() {
        return MyClass;
    }
}

let instance1 = new MyDerivedClass1("foo"),
    clone1 = instance1.clone(),
    instance2 = new MyDerivedClass2("bar"),
    clone2 = instance2.clone();

console.log(clone1 instanceof MyClass);             // true
console.log(clone1 instanceof MyDerivedClass1);     // true
console.log(clone2 instanceof MyClass);             // true
console.log(clone2 instanceof MyDerivedClass2);     // false
```

此处，`MyDerivedClass1` 继承了 `MyClass` ，并且未修改 `Symbol.species` 属性。由于 `this.constructor[Symbol.species]` 会返回 `MyDerivedClass1` ，当 `clone()` 被调用时，它就返回了 `MyDerivedClass1` 的一个实例。 `MyDerivedClass2` 类也继承了 `MyClass` ，但重写了 `Symbol.species` ，让其返回 `MyClass` 。当 `clone()` 在 `MyDerivedClass2` 的一个实例上被调用时，返回值就变成 `MyClass` 的一个实例。使用 `Symbol.species` ，任意派生类在调用应当返回实例的方法时，都可以判断出需要返回什么类型的值。

例如， `Array` 使用了 `Symbol.species` 来指定方法所使用的类，让其返回值为一个数组。在 `Array` 派生出的类中，你可以决定这些继承的方法应返回何种类型的对象，正如：

```
class MyArray extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);      // true
console.log(subitems instanceof Array);     // true
console.log(subitems instanceof MyArray);   // false
```

此代码重写了从 `Array` 派生的 `MyArray` 类上的 `Symbol.species` 。所有返回数组的继承方法现在都会使用 `Array` 的实例，而不是 `MyArray` 的实例。

一般而言，每当想在类方法中使用 `this.constructor` 时，你就应当使用 `Symbol.species` 属性。这么做允许派生类轻易地重写方法的返回类型。此外，若你从一个拥有 `Symbol.species` 定义的类创建了派生类，要保证使用此属性，而不是直接使用构造器。

在类构造器中使用 new.target

在第三章你已学到了 `new.target` ，以及在调用函数的方式不同时它的值是如何变动的。你也可以在类构造器中使用 `new.target` ，来判断类是被如何被调用的。在简单情况下， `new.target` 就等于本类的构造器函数，正如下例；

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

// new.target 就是 Rectangle
var obj = new Rectangle(3, 4);      // 输出 true
```

此代码说明在 `new Rectangle(3, 4)` 被调用时， `new.target` 就等于 `Rectangle` 。类构造器被调用时不能缺少 `new` ，因此 `new.target` 属性就始终会在类构造器内被定义。不过这个值并不总是不变的。研究以下代码：

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

class Square extends Rectangle {
    constructor(length) {
        super(length, length)
    }
}

// new.target 就是 Square
var obj = new Square(3);      // 输出 false
```

`Square` 调用了 `Rectangle` 构造器，因此当 `Rectangle` 构造器被调用时， `new.target` 等于 `Square` 。这很重要，因为这让构造器能根据如何被调用而更改其行为。例如，你可以使用 `new.target` 来创建一个抽象基类（一种不能被实例化的类），如下：

```
// 静态的基类
class Shape {
    constructor() {
        if (new.target === Shape) {
            throw new Error("This class cannot be instantiated directly.")
        }
    }
}

class Rectangle extends Shape {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

var x = new Shape();                // 抛出错误

var y = new Rectangle(3, 4);        // 没有错误
console.log(y instanceof Shape);    // true
```

此例中的 `Shape` 类构造器会在 `new.target` 为 `Shape` 的时候抛出错误，意味着 `new Shape()` 永远都会抛出错误。然而，你依然可以将 `Shape` 用作一个基类，正如 `Rectangle` 所做的那样。 `super()` 的调用执行了 `Shape` 构造器，而且 `new.target` 的值等于 `Rectangle` ，因此该构造器能够无错误地继续执行。

> 由于调用类时不能缺少 `new` ，于是 `new.target` 属性在类构造器内部就绝不会是 `undefined` 。

总结

ES6 的类让 JS 中的继承变得更简单，因此你无须丢弃已从其他语言学习到的类知识。 ES6 的类起初是作为 ES5 传统继承模型的语法糖，但添加了许多特性来减少错误。

ES6 的类配合原型继承来工作，在类的原型上定义了非静态的方法，而静态的方法最终则被绑定在类构造器自身上。类的所有方法从一开始就是不可枚举的，这更契合了内置对象的行为，后者的方法默认情况下通常都不可枚举。此外，调用类构造器时不能缺少 `new` ，确保了类无法意外地被作为函数来调用。

基于类的继承允许你从另一个类、函数或表达式上派生新的类。这种能力意味着你可以调用一个函数来判断需要继承的正确基类，也允许你使用混入或其他不同的组合模式来创建一个新类。新的继承方式让继承数组之类的内置对象也变为可能，并且其工作符合预期。

你可以在类构造器内部使用 `new.target` ，以便根据类如何被调用来做出不同的行为。最常用的就是创建一个抽象基类，直接实例化它会抛出错误，但它仍然能被其他类所继承。

总之，类是 JS 的一项新特性，它提供了更简洁的语法与更好的功能，能通过安全一致的方式来自定义一个对象类型。

第十章 增强的数组功能

数组是 JS 中的一种基本对象。 JS 的其他方面都随着时间的推移在进化，而数组却基本保持不变，直到 ES5 才添加了几个相关的方法让数组更易使用。 ES6 也添加了很多功能来继续强化数组，例如新的创建方法、几个有用的便捷方法，还增加了创建类型化数组（typed array）的能力。

创建数组

在 ES6 之前创建数组主要存在两种方式： `Array` 构造器与数组字面量写法。这两种方式都需要将数组的项分别列出，并且还要受到其他限制。将“类数组对象”（即：拥有数值类型索引与长度属性的对象）转换为数组也并不自由，经常需要书写额外的代码。为了使数组更易创

建，ES6 新增了 `Array.of()` 与 `Array.from()` 方法。

Array.of() 方法

ES6 为数组新增创建方法的目的之一，是帮助开发者在使用 `Array` 构造器时避开 JS 语言的一个怪异点。调用 `new Array()` 构造器时，根据传入参数的类型与数量的不同，实际上会导致一些不同的结果，例如：

```
let items = new Array(2);
console.log(items.length);          // 2
console.log(items[0]);              // undefined
console.log(items[1]);              // undefined

items = new Array("2");
console.log(items.length);          // 1
console.log(items[0]);              // "2"

items = new Array(1, 2);
console.log(items.length);          // 2
console.log(items[0]);              // 1
console.log(items[1]);              // 2

items = new Array(3, "2");
console.log(items.length);          // 2
console.log(items[0]);              // 3
console.log(items[1]);              // "2"
```

当使用单个数值参数来调用 `Array` 构造器时，数组的长度属性会被设置为该参数；而如果使用单个的非数值型参数来调用，该参数就会成为目标数组的唯一项；如果使用多个参数（无论是否为数值类型）来调用，这些参数也会成为目标数组的项。数组的这种行为既混乱又有风险，因为有时可能不会留意所传参数的类型。

ES6 引入了 `Array.of()` 方法来解决这个问题。该方法的作用非常类似 `Array` 构造器，但在使用单个数值参数的时候并不会导致特殊结果。 `Array.of()` 方法总会创建一个包含所有传入参数的数组，而不管参数的数量与类型。下面几个例子演示了 `Array.of()` 的用法：

```
let items = Array.of(1, 2);
console.log(items.length);          // 2
console.log(items[0]);              // 1
console.log(items[1]);              // 2

items = Array.of(2);
console.log(items.length);          // 1
console.log(items[0]);              // 2

items = Array.of("2");
console.log(items.length);          // 1
console.log(items[0]);              // "2"
```

在使用 `Array.of()` 方法创建数组时，只需将想要包含在数组内的值作为参数传入。第一个例子创建了一个包含两个项的数组，第二个数组只包含了单个数值项，而最后一个数组则包含了单个字符串项。这个结果类似于使用数组字面量写法，通常你都可以在原生数组上使用字面量写法来代替 `Array.of()` ，但若想向函数传递参数，使用 `Array.of()` 而非 `Array` 构造器能够确保行为一致。例如：

```
function createArray(arrayCreator, value) {
    return arrayCreator(value);
}

let items = createArray(Array.of, value);
```

此代码中的 `createArray()` 函数接受两个参数：一个数组创建器与一个值，并会将后者插入到目标数组中。你应当向 `createArray()` 函数传递 `Array.of()` 作为第一个参数来创建新数组；相反，若传递 `Array` 构造器则会有危险，因为你无法保证第二个参数不是数值类型。

> `Array.of()` 方法并没有使用 `Symbol.species` 属性（参阅第九章）来决定返回值的类型，而是使用了当前的构造器（即 `of()` 方法内部的 `this` ）来做决定。

Array.from() 方法

在 JS 中将非数组对象转换为真正的数组总是很麻烦。例如，若想将类数组的 `arguments` 对象当做数组来使用，那么你首先需要对其进行转换。在 ES5 中，进行这种转换需要编写一个函数，类似下面这样：

```
function makeArray(arrayLike) {
    var result = [];

    for (var i = 0, len = arrayLike.length; i < len; i++) {
        result.push(arrayLike[i]);
    }

    return result;
}

function doSomething() {
    var args = makeArray(arguments);

    // 使用 args
}
```

该方式手动创建了一个 `result` 数组，并将 `arguments` 对象的所有项复制到该数组中。这种方式虽然有效，却为一个简单操作书写了过多的代码。开发者最终发现他们可以调用数组原生的 `slice()` 方法来减少代码量，就像这样：

```
function makeArray(arrayLike) {
    return Array.prototype.slice.call(arrayLike);
}

function doSomething() {
    var args = makeArray(arguments);

    // 使用 args
}
```

这段代码的功能与前一段代码等效。它能正常工作是因为将 `slice()` 方法的 `this` 设置为类数组对象， `slice()` 只需要有数值类型的索引与长度属性就能正常工作，而类数组对象能满足这些要求。

尽管这种技巧所用的代码量更少，但调用 `Array.prototype.slice.call(arrayLike)` 并没有明确体现出“要将类数组对象转换为数组”的目的。幸运的是， ES6 新增了 `Array.from()` 方法来提供一种明确清晰的方式以解决这方面的需求。

将可迭代对象或者类数组对象作为第一个参数传入， `Array.from()` 就能返回一个数组。此处有个简单的例子：

```
function doSomething() {
    var args = Array.from(arguments);

    // 使用 args
}
```

此处调用 `Array.from()` 方法，使用 `arguments` 对象创建了一个新数组 `args` ，它是一个数组实例，并且包含了 `arguments` 对象的所有项，同时还保持了项的顺序。

> `Array.from()` 方法同样使用 `this` 来决定要返回什么类型的数组。

映射转换

如果你想实行进一步的数组转换，可以向 `Array.from()` 方法传递一个映射用的函数作为第二个参数。此函数会将类数组对象的每一个值转换为目标形式，并将其存储在目标数组的对应位置上。例如：

```
function translate() {
    return Array.from(arguments, (value) => value + 1);
}

let numbers = translate(1, 2, 3);

console.log(numbers);               // 2,3,4
```

此代码将 `(value) => value + 1` 作为映射函数传递给了 `Array.from()` 方法，对每个项进行了一次 +1 处理。如果映射函数需要在对象上工作，你可以手动传递第三个参数给 `Array.from()` 方法，从而指定映射函数内部的 `this` 值。

```
let helper = {
    diff: 1,

    add(value) {
        return value + this.diff;
    }
};

function translate() {
    return Array.from(arguments, helper.add, helper);
}

let numbers = translate(1, 2, 3);

console.log(numbers);               // 2,3,4
```

这个例子使用了 `helper.add()` 作为映射函数。由于该函数使用了 `this.diff` 属性，你必须向 `Array.from()` 方法传递第三个参用于指定 `this` 。借助这个参数， `Array.from()` 就可以方便地进行数据转换，而无须调用 `bind()` 方法、或用其他方式去指定 `this` 值。

在可迭代对象上使用

`Array.from()` 方法不仅可用于类数组对象，也可用于可迭代对象，这意味着该方法可以将任意包含 `Symbol.iterator` 属性的对象转换为数组。例如：

```
let numbers = {
    *[Symbol.iterator]() {
        yield 1;
        yield 2;
        yield 3;
    }
};

let numbers2 = Array.from(numbers, (value) => value + 1);

console.log(numbers2);              // 2,3,4
```

由于代码中的 `numbers` 对象是一个可迭代对象，你可以把它直接传递给 `Array.from()` 方法，从而将它包含的值转换到数组内。映射函数对每个数都进行了 +1 处理，因此目标数组的内容就是 2 、 3 、 4 ，而不是 1 、 2 、 3 。

> 如果一个对象既是类数组对象，又是可迭代对象，那么 `Array.from()` 方法就会使用迭代器来决定需要转换的值。

所有数组上的新方法

ES6 延续了 ES5 的工作，为数组增加了几个新方法。 `find()` 与 `findIndex()` 方法是为了让开发者能够处理包含任意值的数组，而 `fill()` 与 `copyWithin()` 方法则是受到了类型化数组（ **typed arrays** ）的启发。类型化数组是在 ES6 中引入的，只允许包含数值类型的值。

find() 与 findIndex() 方法

在 ES5 之前，由于没有对应的原生方法，检索数组相当麻烦。 ES5 增加了 `indexOf()` 与 `lastIndexOf()` 方法，从而允许开发者在数组中查找特定值。这虽然是很大的进步，但依然有局限性，因为你每次只能用它们来查找某个特定值。例如，若想在一系列数值中间查找第一个偶数，你必须自己写代码来实现这个意图。而 ES6 引入了 `find()` 与 `findIndex()` 方法，从而解决了这方面的问题。

`find()` 与 `findIndex()` 方法均接受两个参数：一个回调函数、一个可选值用于指定回调函数内部的 `this` 。该回调函数可接收三个参数：数组的某个元素、该元素对应的索引位置、以及该数组自身，这与 `map()` 和 `forEach()` 方法的回调函数所用的参数一致。在给定的元素满足你定义的条件时，回调函数应当返回 `true` ，而 `find()` 与 `findIndex()` 方法均会在回调函数第一次返回 `true` 时停止查找。

二者唯一的区别是： `find()` 方法会返回匹配的值，而 `findIndex()` 方法则会返回匹配位置的索引。此处有个示例：

```
let numbers = [25, 30, 35, 40, 45];

console.log(numbers.find(n => n > 33));         // 35
console.log(numbers.findIndex(n => n > 33));    // 2
```

这段代码使用了 `find()` 与 `findIndex()` 方法在 `numbers` 数组中查找第一个大于 33 的元素，前者返回 35 ，后者则返回 2 ，也就是 35 这个元素在数组中的索引值。

`find()` 与 `findIndex()` 方法在查找满足特定条件的数组元素时非常有用。但若只想查找特定值， `indexOf()` 与 `lastIndexOf()` 方法则是更好的选择。

fill() 方法

`fill()` 方法能使用特定值填充数组中的一个或多个元素。当只使用一个参数的时候，该方法会用该参数的值填充整个数组，例如：

```
let numbers = [1, 2, 3, 4];

numbers.fill(1);

console.log(numbers.toString());    // 1,1,1,1
```

此代码中的 `numbers.fill(1)` 调用将 `numbers` 数组中的所有元素都填充为 `1` 。若你不想改变数组中的所有元素，而只想改变其中一部分，那么可以使用可选的起始位置参数与结束位置参数（不包括结束位置的那个元素），就像这样：

```
let numbers = [1, 2, 3, 4];

numbers.fill(1, 2);

console.log(numbers.toString());    // 1,2,1,1

numbers.fill(0, 1, 3);

console.log(numbers.toString());    // 1,0,0,1
```

当进行 `numbers.fill(1,2)` 调用时，第二个参数 `2` 指定从数组索引值为 2 的元素（即数组的第 3 个元素）开始填充，而此时没有指定第三个参数，因此结束位置默认为 `numbers` 数组的长度，意味着该数组的最后两个元素会被填充为 `1` 。而 `numbers.fill(0, 1, 3)` 调用则将该数组索引值为 1 与 2 的元素填充为 `0` 。在调用 `fill()` 方法时指定第二个和第三个参数，允许一次性填充数组中多个元素，避免改写整个数组。

> 如果提供的起始位置或结束位置为负数，则它们会被加上数组的长度来算出最终的位置。例如：将起始位置指定为 `-1` ，就等于是 `array.length - 1` ，此处的 `array` 指的是 `fill()` 方法所要处理的数组。

copyWithin() 方法

`copyWithin()` 方法与 `fill()` 类似，也能一次性修改数组的多个元素。不过，与 `fill()` 使用单个值来填充数组不同， `copyWithin()` 方法允许你在数组内部复制自身元素。为此你需要传递两个参数给 `copyWithin()` 方法：从什么位置开始进行填充，以及被用来复制的数据的起始位置索引。

例如，将数组的前两个元素复制到数组的最后两个位置，你可以这么做：

```
let numbers = [1, 2, 3, 4];

// 从索引 2 的位置开始粘贴
// 从数组索引 0 的位置开始复制数据
numbers.copyWithin(2, 0);

console.log(numbers.toString());    // 1,2,1,2
```

这段代码从 `numbers` 数组索引值为 2 的元素开始进行填充，因此索引值为 2 与 3 的元素都会被覆盖；调用 `copyWithin()` 方法时将第二个参数指定为 `0` ，表示被复制的数据从索引值为 0 的元素开始，一直到没有元素可供复制为止。

默认情况下， `copyWithin()` 方法总是会一直复制到数组末尾，不过你还可以提供一个可选参数来限制到底有多少元素会被覆盖。这第三个参数指定了复制停止的位置（不包含该位置自身），此处有个范例：

```
let numbers = [1, 2, 3, 4];

// 从索引 2 的位置开始粘贴
// 从数组索引 0 的位置开始复制数据
// 在遇到索引 1 时停止复制
numbers.copyWithin(2, 0, 1);

console.log(numbers.toString());    // 1,2,1,4
```

在这个例子中，因为可选的结束位置参数被指定为 `1` ，于是只有索引值为 0 的元素被复制了，而该数组的最后一个元素并没有被修改。

> 类似于 `fill()` 方法，如果你向 `copyWithin()` 方法传递负数参数，数组的长度会自动被加到该参数的值上，以便算出正确的索引位置。

`fill()` 与 `copyWithin()` 方法乍看不太有用，因为它们起源于类型化数组的需求，而出于功能一致性的目的才被添加到常规数组上。不过，接下来的小节你就会学到如何用类型化数组来按位操作数值，到时这两个方法就会大显身手。

类型化数组

类型化数组是有特殊用途的数组，被设计用来处理数值类型数据（正如名称所示，不是所有类型）。类型化数组的起源可以追溯到 WebGL —— Open GL ES 2.0 的一个接口，设计用于配合网页上的 `<canvas>` 元素。类型化数组作为该接口实现的一部分，为 JS 提供了快速的按位运算能力。

对于 WebGL 的需求来说， JS 原生的数学运算实在太慢，因为它使用 64 位浮点数格式来存储数值，并在必要时将其转换为 32 位整数。引入类型化数组突破了格式限制并带来了更好的数学运算性能，其设计概念是：单个数值可以被视为由“位”构成的数组，并且可以对其使用与 JS 数组现有方法类似的方法。

ES6 采纳了类型化数组，将其作为语言的一个正式部分，以确保在 JS 引擎之间有更好的兼容性，并确保与 JS 数组有更好的互操作性。尽管 ES6 的类型化数组与 WebGL 的类型化数组并不完全一样，但它们已足够相似，让前者能被视为后者的进化版本，而非截然不同。

数值数据类型

JS 数值使用 IEEE 754 标准格式存储，使用 64 位来存储一个数值的浮点数表示形式，该格式在 JS 中被同时用来表示整数与浮点数；当值改变时，可能会频繁发生整数与浮点数之间的格式转换。而类型化数组则允许存储并操作八种不同的数值类型：

1. 8 位有符号整数（int8）
2. 8 位无符号整数（uint8）
3. 16 位有符号整数（int16）
4. 16 位无符号整数（uint16）
5. 32 位有符号整数（int32）
6. 32 位无符号整数（uint32）
7. 32 位浮点数（float32）
8. 64 位浮点数（float64）

如果你将一个 int8 范围内的数表示为常规的 JS 数值，你就浪费了 56 个位，而这些位原本可用来存储额外的 int8 值、或任意需求小于 56 位的数值。更有效地利用“位”是类型化数组的处理用途之一。

所有与类型化数组相关的操作和对象都围绕着这八种数据类型。为了使用它们，你首先需要创建一个数组缓冲区用于存储数据。

> 在本书中，我将使用上述列表中括号内的缩写词来表示这些类型，不过这些缩写并不会出现在实际的 JS 代码中，因为它们仅仅是对超长描述信息的速记。

数组缓冲区

数组缓冲区（**array buffer**）是内存中包含一定数量字节的区域，而所有的类型化数组都基于数组缓冲区。创建数组缓冲区类似于在 C 语言中使用 `malloc()` 来分配内存，而不需要指定这块内存包含什么。你可以像下例这样使用 `ArrayBuffer` 构造器来创建一个数组缓冲区：

```
let buffer = new ArrayBuffer(10);   // 分配了 10 个字节
```

调用 `ArrayBuffer` 构造器时，只需要传入单个数值用于指定缓冲区包含的字节数，而本例就创建了一个 10 字节的缓冲区。当数组缓冲区被创建完毕后，你就可以通过检查 `byteLength` 属性来获取缓冲区的字节数：

```
let buffer = new ArrayBuffer(10);   // 分配了 10 个字节
console.log(buffer.byteLength);     // 10
```

你还可以使用 `slice()` 方法来创建一个包含已有缓冲区部分内容的新数组缓冲区。该 `slice()` 方法类似于数组上的同名方法，可以使用起始位置与结束位置参数，返回由原缓冲区元素组成的一个新的 `ArrayBuffer` 实例。例如：

```
let buffer = new ArrayBuffer(10);   // 分配了 10 个字节


let buffer2 = buffer.slice(4, 6);
console.log(buffer2.byteLength);    // 2
```

此代码创建了 `buffer2` 数组缓冲区，提取了原缓冲区索引值为 4 与 5 的元素。与数组的同名方法一样，结束参数所对应的元素是不会包含在结果中的。

当然，仅仅创建一个存储区域而不能写入数据，没有什么意义。你需要创建一个视图（view）来进行写入：

> 数组缓冲区总是保持创建时指定的字节数，你可以修改其内部的数据，但永远不能修改它的容量。

使用视图操作数组缓冲区

数组缓冲区代表了一块内存区域，而视图（**views**）则是你操作这块区域的接口。视图工作在数组缓冲区或其子集上，可以读写某种数值数据类型的数据。 `DataView` 类型是数组缓冲区的通用视图，允许你对前述所有八种数值数据类型进行操作。

使用 `DataView` ，首先需要创建 `ArrayBuffer` 的一个实例，再在上面创建一个新的 `ArrayBuffer` 视图。此处有个例子：

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer);
```

本例中的 `view` 对象可以使用 `buffer` 对象的所有 10 个字节。而你也可以在缓冲区的一个部分上创建视图，只需要指定可选参数——字节偏移量、以及所要包含的字节数。当未提供最后一个参数时，该 `DataView` 视图会默认包含从偏移位置开始、到缓冲区末尾为止的元素。例如：

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer, 5, 2);      // 包含位置 5 与位置 6 的字节
```

此例中的 `view` 只能使用索引值为 5 与 6 的字节。使用这种方式，你可以在同一个数组缓冲区上创建多个不同的视图，这样有助于将单块内存区域供给整个应用使用，而不必每次在有需要时才动态分配内存。

获取视图信息

你可以通过查询以下只读属性来获取视图的信息：

- `buffer` ：该视图所绑定的数组缓冲区；
- `byteOffset` ：传给 `DataView` 构造器的第二个参数，如果当时提供了的话（默认值为 0）；
- `byteLength` ：传给 `DataView` 构造器的第三个参数，如果当时提供了的话（默认值为该缓冲区的 `byteLength` 属性）。

使用这些属性，你就可以查出所操作视图的准确位置，例如：

```
let buffer = new ArrayBuffer(10),
    view1 = new DataView(buffer),           // 包含所有字节
    view2 = new DataView(buffer, 5, 2);     // 包含位置 5 与位置 6 的字节

console.log(view1.buffer === buffer);       // true
console.log(view2.buffer === buffer);       // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

此代码创建了包含整个缓冲区的 `view1` 视图，并创建了包含缓冲区一小部分的 `view2` 视图。这两个视图拥有相同的 `buffer` 属性值，因为它们是在同一个数组缓冲区上工作的；而二者的 `byteOffset` 与 `byteLength` 属性就不相等了，这些属性反映出视图使用了缓冲区的哪

些部分。

当然，仅仅读取缓冲区的内存信息还不够，你还需要能向其写入数据并重新读出数据。

读取与写入数据

对应于 JS 所有八种数值数据类型， `DataView` 视图的原型分别提供了在数组缓冲区上写入与读取数据的方法。所有方法名都以“set”或“get”开始，其后跟随着对应数据类型的缩写。下面列出了能够操作 int8 或 uint8 类型的读取/写入方法：

- `getInt8(byteOffset, littleEndian)` ：从 `byteOffset` 处开始读取一个 int8 值；
- `setInt8(byteOffset, value, littleEndian)` ：从 `byteOffset` 处开始写入一个 int8 值；
- `getUint8(byteOffset, littleEndian)` ：从 `byteOffset` 处开始读取一个 uint8 值；
- `setUint8(byteOffset, value, littleEndian)` ：从 `byteOffset` 处开始写入一个 uint8 值。

“get”方法接受两个参数：开始进行读取的字节偏移量、以及一个可选的布尔值，后者用于指定读取的值是否采用低字节优先方式（注：默认值为 `false` ）。“set”方法则接受三个参数：开始进行写入的字节偏移量、需要写入的数据值、以及一个可选的布尔值用于指定是否采用低字节优先方式存储数据。

> 译注：低字节优先（**Little-endian**）也被翻译作“小端字节序”，指的是在存储数据的多个内存字节中，第一个内存字节存储着数据的最低字节数据，而最后一个内存字节存储着最高字节数据。
>
> 例如：十进制数 5882 用十六进制表示是 16FA ，如果采用低字节优先方式、并使用 4 字节（即 32 位）存储，则该数字在内存中会被存储为 FA 16 00 00 。而如果采用相反的存储方式：高字节优先（**Big-endian**，大端字节序），那么该数字则会被存储为 00 00 16 FA 。

尽管上面只列出了操作 8 位值的方法，但只要将方法名中的 `8` 替换为 `16` 或 `32` ，便可以用来操作 16 位或 32 位值。而除了这些整数类方法之外， `DataView` 也提供了下列读写方法以便处理浮点数：

- `getFloat32(byteOffset, littleEndian)` ：从 `byteOffset` 处开始读取一个 float32 值；
- `setFloat32(byteOffset, value, littleEndian)` ：从 `byteOffset` 处开始写入一个 float32 值；
- `getFloat64(byteOffset, littleEndian)` ：从 `byteOffset` 处开始读取一个 float64 值；
- `setFloat64(byteOffset, value, littleEndian)` ：从 `byteOffset` 处开始写入一个 float64 值。

为了弄明白“set”与“get”方法如何使用，可研究下面的例子：

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt8(0));       // 5
console.log(view.getInt8(1));       // -1
```

该代码使用一个双字节的数组缓冲区来存储两个 int8 值。第一个值被存储在位置 0 ，而第二个值则被存储在位置 1 ，表示每个值占用了一个完整的字节（8 位），此后还使用 `getInt8()` 方法来将这些值从对应位置读取出来。尽管这个例子只使用了 int8 类型的值，但你却可以使用八种数值数据类型的所有对应方法。

视图允许你使用任意格式对任意位置进行读写，而无须考虑这些数据此前是使用什么格式存储的，这非常有意思。例如，完全可以向缓冲区写入两个 int8 值，并将其作为一个 int16 值读取出来，如同下面这个例子：

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt16(0));      // 1535
console.log(view.getInt8(0));       // 5
console.log(view.getInt8(1));       // -1
```

该代码使用 `view.getInt16(0)` 读取了该视图的所有字节，并将其解析为数值 1535 。为了理解这个范例，可以参阅下面的示意图，它揭示了每个 `setInt8()` 操作对缓冲区造成的变化：

```
new ArrayBuffer(2)      0000000000000000
view.setInt8(0, 5);     0000010100000000
view.setInt8(1, -1);    0000010111111111
```

开始时，该数组缓冲区 16 个位均为 0 ；使用 `setInt8()` 向第一个字节写入 `5` 之后，该字节的内容就出现了一对 1 （因为 5 可以写为 8 位二进制数 00000101 ）；向第二个字节写入 -1 会使得该字节的所有位都变成 1 （即 -1 的二进制补码形式）。接下来使用 `getInt16()` 就能将前面写入的 16 位数据以单个 16 位整数的方式读取出来，其十进制值就是 1535 。

在混用不同的数据类型时，使用 `DataView` 对象是一种完美方式。不过，若仅想使用特定的一种数据类型，那么特定类型视图会是更好的选择。

类型化数组即为视图

ES6 的类型化数组实际上也是针对数组缓冲区的特定类型视图，你可以使用这些数组对象来处理特定的数据类型，而不必使用通用的 `DataView` 对象。一共存在八种特定类型视图，对应着八种数值数据类型，还为处理 `uint8` 值提供了额外的选择。

特定类型视图被包含在 ES6 规范的 22.2 小节中，下表列出了它们的概要：

构造器名称	元素大小（字节）	描述	等价的 C 语言类型
`Int8Array`	1	8 位有符号整数，采用补码	`signed char`
`Uint8Array`	1	8 位无符号整数	`unsigned char`
`Uint8ClampedArray`	1	8 位无符号整数 (clamped conversion，无溢出转换)	`unsigned char`
`Int16Array`	2	16 位有符号整数，采用补码	`short`
`Uint16Array`	2	16 位无符号整数	`unsigned short`
`Int32Array`	4	32 位有符号整数，采用补码	`int`
`Uint32Array`	4	32 位无符号整数	`int`
`Float32Array`	4	32 位 IEEE 浮点数	`float`
`Float64Array`	8	64 位 IEEE 浮点数	`double`

左边一列列出了类型化数组的构造器，而其他列则描述了对应的类型化数组所能包含的数据。 `Uint8ClampedArray` 的特性与 `Uint8Array` 基本相同，只有当缓冲区包含的值小于 0 或者大于 255 的时候才有区别：当值小于 0 时， `Uint8ClampedArray` 会将该值转换为 0 进行存储（例如 -1 会被存储为 0 ）；而当值大于 255 时，会被转换为 255 （例如 300 会被存储为 255 ）。

类型化数组只能在特定的一种数据类型上工作，例如： `Int8Array` 的所有操作都只能处理 `int8` 值。每种类型化数组的单个元素大小也都取决于对应类型， `Int8Array` 中每个元素都是单字节的，而 `Float64Array` 则使用了八个字节来存储单个元素。幸运的是，类型化数组的元素可以使用数值型的索引位置来访问，就像常规数组那样，从而规避了使用 `DataView` 存取方法时的某些尴尬情况。

> 元素大小
>
> 每一种类型化数组都由一定数量的元素构成，而“元素大小”则代表每个类型的单个元素所包含的字节数。这个数字被存储在类型化数组每个构造器与每个实例的 `BYTES_PER_ELEMENT` 属性中，方便你查询元素的大小：
>
> ```
> console.log(UInt8Array.BYTES_PER_ELEMENT);      // 1
> console.log(UInt16Array.BYTES_PER_ELEMENT);     // 2
>
> let ints = new Int8Array(5);
> console.log(ints.BYTES_PER_ELEMENT);            // 1
> ```

创建特定类型视图

类型化数组的构造器可以接受多种类型的参数，因此存在几种创建类型化数组的方式。第一种方式是使用与创建 `DataView` 时相同的参数，即：一个数组缓冲区、一个可选的字节偏移量、以及一个可选的字节数量。例如：

```
let buffer = new ArrayBuffer(10),
    view1 = new Int8Array(buffer),
    view2 = new Int8Array(buffer, 5, 2);

console.log(view1.buffer === buffer);       // true
console.log(view2.buffer === buffer);       // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

此代码在 `buffer` 对象上创建了两个 `Int8Array` 类型的视图： `view1` 与 `view2` ，而这两个视图拥有相同的 `buffer` 、 `byteOffset` 与 `byteLength` 属性。如果你的操作只针对一种数值类型，那么很容易就能把代码从使用 `DataView` 视图切换到使用某种类型化数组。

第二种方式是传递单个数值给类型化数组的构造器，此数值表示该数组包含的元素数量（而不是分配的字节数）。构造器将会创建一个新的缓冲区，分配正确的字节数以便容纳指定数量的数组元素，而你也可以使用 `length` 属性来获取这个元素数量。例如：

```
let ints = new Int16Array(2),
    floats = new Float32Array(5);

console.log(ints.byteLength);       // 4
console.log(ints.length);           // 2

console.log(floats.byteLength);     // 20
console.log(floats.length);         // 5
```

示例中的 `ints` 数组创建时包含了两个元素，而每个 16 位整数需要使用两个字节，因此该数组一共被分配了 4 个字节。 `floats` 数组则包含五个元素，每个元素占用四个字节，因此它就需要 20 个字节。这两个数组都创建了对应的数组缓冲区，而在必要时都可以使用 `buffer` 属性来访问各自的缓冲区。

> 如果调用类型化数组构造器时没有传入参数，构造器会认为传入了 `0` ，这种方式创建的类型化数组不会被分配任何存储空间，因此也就不能被用于保存数据。

第三种方式是向构造器传递单个对象参数，可以是下列四种对象之一：

- 类型化数组：数组所有元素都会被复制到新的类型化数组中。例如，如果你传递一个 int8 类型的数组给 `Int16Array` 构造器，这些 int8 的值会被复制到 int16 数组中。新的类型化数组与原先的类型化数组会使用不同的数组缓冲区。
- 可迭代对象：该对象的迭代器会被调用以便将数据插入到类型化数组中。如果其中包含了不匹配视图类型的值，那么构造器就会抛出错误。
- 数组：该数组的元素会被插入到新的类型化数组中。如果其中包含了不匹配视图类型的值，那么构造器就会抛出错误。
- 类数组对象：与传入数组的表现一致。

在上述任意可能中，新的类型化数组都会从原对象获取数据。若想用一些值来初始化一个类型化数组，这种方式就特别有用，就像这样：

```
let ints1 = new Int16Array([25, 50]),
    ints2 = new Int32Array(ints1);

console.log(ints1.buffer === ints2.buffer);     // false

console.log(ints1.byteLength);      // 4
console.log(ints1.length);          // 2
console.log(ints1[0]);              // 25
console.log(ints1[1]);              // 50

console.log(ints2.byteLength);      // 8
console.log(ints2.length);          // 2
console.log(ints2[0]);              // 25
console.log(ints2[1]);              // 50
```

该例使用了一个包含两个值的数组来创建一个 `Int16Array` 并初始化它，之后又利用该 `Int16Array` 创建了一个 `Int32Array` 。25 与 50 这两个值从 `ints1` 数组中被复制到 `ints2` 数组中，但两个数组使用了全然不同的缓冲区。虽然二者都包含了相同的数值，但后者占用了 8 个字节，而前者只占用了 4 个字节。

类型化数组与常规数组的相似点

类型化数组与常规数组有好几个相似点，并且正如你已经在本章看到的那样，在很多场景中都可以像使用常规数组那样使用类型化数组。例如，你可以使用 `length` 属性来获取类型化数组包含的元素数量，还可以使用数值类型的索引值来直接访问类型化数组的元素。举个例子：

```
let ints = new Int16Array([25, 50]);

console.log(ints.length);          // 2
console.log(ints[0]);              // 25
console.log(ints[1]);              // 50

ints[0] = 1;
ints[1] = 2;

console.log(ints[0]);              // 1
console.log(ints[1]);              // 2
```

这段代码创建了一个包含两个元素的 `Int16Array` ，使用数值类型的索引可以读写对应的项，而数值在存储时会被自动转换为 int16 类型的值。

相似点还不限于此。

> 与常规数组不同的是，你不能使用 `length` 属性来改变类型化数组的大小。该属性是不可写的，在非严格模式下写入操作会被忽略，而严格模式下则会抛出错误。

公共方法

类型化数组也拥有大量与常规数组等效的方法，你可以对类型化数组使用下列这些方法：

- `copyWithin()`
- `entries()`
- `fill()`
- `filter()`
- `find()`
- `findIndex()`
- `forEach()`
- `indexOf()`
- `join()`
- `keys()`
- `lastIndexOf()`
- `map()`
- `reduce()`
- `reduceRight()`
- `reverse()`

- `slice()`
- `some()`
- `sort()`
- `values()`

注意：虽然这些方法与数组原型上的对应方法表现相似，但它们并不完全相同。类型化数组的方法会进行额外的类型检查以确保安全，并且返回值会根据 `Symbol.species` 属性来确定，会是某种类型化数组而非常规数组。此处有个例子用于演示其中的区别：

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => v * 2);

console.log(mapped.length);         // 2
console.log(mapped[0]);             // 50
console.log(mapped[1]);             // 100

console.log(mapped instanceof Int16Array);  // true
```

这段代码通过 `map()` 方法使用 `ints` 中的值创建了一个新数组，映射函数将每个值翻倍，并返回了一个新的 `Int16Array` 。

相同的迭代器

与常规数组相同，类型化数组也拥有三个迭代器，它们是 `entries()` 方法、 `keys()` 方法与 `values()` 方法。这就意味着你可以对类型化数组使用扩展运算符，或者对其使用 `for-of` 循环，就像对待常规数组。举个例子：

```
let ints = new Int16Array([25, 50]),
    intsArray = [...ints];

console.log(intsArray instanceof Array);    // true
console.log(intsArray[0]);                  // 25
console.log(intsArray[1]);                  // 50
```

此代码创建了一个名为 `intsArray` 的新数组，包含了类型化数组 `ints` 的所有数据。借助扩展运算符能轻易地将类型化数组转换为常规数组，就像处理其他可迭代对象那样。

of() 与 from() 方法

最后，所有的类型化数组都包含静态的 `of()` 与 `from()` 方法，作用类似于 `Array.of()` 与 `Array.from()` 方法。其中的区别是类型化数组的版本会返回类型化数组，而不返回常规数组。下面的例子使用这两个方法创建了几个类型化数组：

```
let ints = Int16Array.of(25, 50),
    floats = Float32Array.from([1.5, 2.5]);

console.log(ints instanceof Int16Array);        // true
console.log(floats instanceof Float32Array);    // true

console.log(ints.length);         // 2
console.log(ints[0]);             // 25
console.log(ints[1]);             // 50

console.log(floats.length);       // 2
console.log(floats[0]);           // 1.5
console.log(floats[1]);           // 2.5
```

此例中分别使用了 `of()` 与 `from()` 方法来创建一个 `Int16Array` 以及一个 `Float32Array` ，这两个方法确保创建类型化数组能像创建常规数组那样轻松。

类型化数组与常规数组的区别

二者最重要的区别就是类型化数组并不是常规数组，类型化数组并不是从 `Array` 对象派生的，使用 `Array.isArray()` 去检测会返回 `false` ，例如：

```
let ints = new Int16Array([25, 50]);

console.log(ints instanceof Array);       // false
console.log(Array.isArray(ints));         // false
```

由于 `ints` 变量是一个类型化数组，因此它并不是 `Array` 对象的实例，于是就不会被识别为数组。这一点区别很重要，因为虽然类型化数组与常规数组非常相似，但前者仍然有一些不同的行为。

行为差异

常规数组可以被伸展或是收缩，然而类型化数组却会始终保持自身大小不变。你可以对常规数组一个不存在的索引位置进行赋值，但在类型化数组上这么做则会被忽略。此处有个例子：

```
let ints = new Int16Array([25, 50]);

console.log(ints.length);          // 2
console.log(ints[0]);              // 25
console.log(ints[1]);              // 50

ints[2] = 5;

console.log(ints.length);          // 2
console.log(ints[2]);              // undefined
```

在本例中，尽管对索引值 2 的位置进行了赋值为 `5` 的操作，但 `ints` 数组却完全没有被伸展，数组的长度属性保持不变，所赋的值也被丢弃了。

类型化数组也会对数据类型进行检查以保证只使用有效的值，当无效的值被传入时，将会被替换为 0 ，例如：

```
let ints = new Int16Array(["hi"]);

console.log(ints.length);       // 1
console.log(ints[0]);           // 0
```

这段代码试图用字符串值 `"hi"` 创建一个 `Int16Array` ，而字符串对于类型化数组来说当然是无效的值，因此该字符串被替换为 `0` 并插入数组。此数组的长度仅仅是 1 ，而 `ints[0]` 只包含了 `0` 这个值。

所有在类型化数组上修改项目值的方法都会受到相同的限制，例如当 `map()` 方法使用的映射函数返回一个无效值的时候，类型化数组会使用 `0` 来代替返回值：

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => "hi");

console.log(mapped.length);        // 2
console.log(mapped[0]);            // 0
console.log(mapped[1]);            // 0

console.log(mapped instanceof Int16Array);  // true
console.log(mapped instanceof Array);       // false
```

由于字符串值 `"hi"` 并不是一个 16 位整数，它在结果数组中就被替换成为 `0` 。多亏这种纠错行为，类型化数组的内容永远不会是无效值，因此相关方法就无须再担心传入无效值会导致错误。

遗漏的方法

尽管类型化数组拥有常规数组的很多同名方法，但仍然缺少了几个数组方法，包括下列这些：

- `concat()`
- `pop()`
- `push()`
- `shift()`
- `splice()`
- `unshift()`

除了 `concat()` 方法之外，该列表中的其余方法都会改变数组的大小，而由于类型化数组的大小不可变，因此这些方法都不能作用于类型化数组。 `concat()` 方法不可用的原因则是：连接两个类型化数组的结果是不确定的（特别是当它们处理的数据类型不同时），这种不确定情况原本就不应当使用类型化数组。

附加的方法

最后，类型化数组还有两个常规数组所不具备的方法： `set()` 方法与 `subarray()` 方法。这两个方法作用相反： `set()` 方法从其他数组中复制元素到当前的类型化数组，而 `subarray()` 方法则是将当前类型化数组的一部分提取为新的类型化数组。

`set()` 方法接受一个数组参数（无论是类型化的还是常规的）、以及一个可选的偏移量参数，后者指示了从什么位置开始插入数据（默认值为 0 ）。数组参数中的数据会被复制到目标类型化数组中，并会确保数据值有效。此处有个例子：

```
let ints = new Int16Array(4);

ints.set([25, 50]);
ints.set([75, 100], 2);

console.log(ints.toString());   // 25,50,75,100
```

这段代码创建了一个包含四个元素的 `Int16Array` ；第一次调用 `set()` 复制了两个值到数组起始的两个位置；而第二次调用 `set()` 则使用了一个值为 `2` 的偏移量参数，指明应当从数组的第三个位置（索引 2 ）开始放置所复制的数据。

`subarray()` 方法接受一个可选的开始位置索引参数、以及一个可选的结束位置索引参数（像 `slice()` 方法一样，结束位置的元素不会被包含在结果中），并会返回一个新的类型化数组。你可以同时省略这两个参数，从而创建原类型化数组的一个复制品。例如：

```
let ints = new Int16Array([25, 50, 75, 100]),
    subints1 = ints.subarray(),
    subints2 = ints.subarray(2),
    subints3 = ints.subarray(1, 3);

console.log(subints1.toString());   // 25,50,75,100
console.log(subints2.toString());   // 75,100
console.log(subints3.toString());   // 50,75
```

本例中利用 `ints` 数组创建了三个类型化数组。 `subints1` 数组是 `ints` 的一个复制品，包含了原数组的所有信息；而 `subints2` 则从原数组的索引 2 位置开始复制，因此包含了原数组的最末两个元素（即 75 与 100 ）；最后的 `subints3` 数组值包含了原数组的中间两个元素，因为调用 `subarray()` 时同时使用了起始位置与结束位置参数。

总结

ES6 延续了 ES5 的工作以便让数组更加有用。新增了两种创建数组的方式： `Array.of()` 方法、以及 `Array.from()` 方法，后者可以将可迭代对象或类数组对象转换为正规数组。这两个方法都在数组派生对象上被继承，并使用 `Symbol.species` 属性来决定返回的数组类型，而其他的继承方法在返回数组时也会使用该属性。

此外还有几个新增的数组方法。 `fill()` 方法与 `copyWithin()` 方法允许你替换数组内的元素。 `find()` 方法与 `findIndex()` 方法在数组中查找满足特定条件的元素时会非常有用，其中前者会返回满足条件的第一个元素，而后者会返回该元素的索引位置。

严格来说类型化数组并不是数组，它们并没有继承 `Array` 对象，但它们的外观和行为都与数组有许多相似点。类型化数组包含的数据类型是八种数值数据类型之一，基于数组缓冲区对象建立，用于表示按位存储的一个数值或一系列数值。类型化数组能够明显提升按位运算的性能，因为它不像 JS 的常规数值类型那样需要频繁进行格式转换。

第十一章 Promise与异步编程

JS 最强大的一方面就是它能极其轻易地处理异步编程。作为因互联网而生的语言， JS 从一开始就必须能够响应点击或按键之类的用户交互行为。 Node.js 通过使用回调函数来代替事件，进一步推动了 JS 中的异步编程。随着越来越多的程序开始使用异步编程，事件与回调函数已不足以支持开发者的所有需求。 **Promise** 正是为了解决这方面的问题。

Promise 是异步编程的另一种选择，它的工作方式类似于在其他语言中进行延迟并在将来执行作业。一个 Promise 指定一些稍后要执行的代码（就像事件与回调函数一样），并且也明确标示了作业的代码是否执行成功。你能以成功处理或失败处理为基准，将 Promise 串联在一起，让代码更易理解、更易调试。

不过为了更好理解 Promise 是如何工作的，重要的是理解建立它所依据的一些基本概念。

异步编程的背景

JS 引擎建立在单线程事件循环的概念上。单线程（ **Single-threaded** ）意味着同一时刻只能执行一段代码，与 Java 或 C++ 这种允许同时执行多段不同代码的多线程语言形成了反差。多段代码可以同时访问或修改状态，维护并保护这些状态就变成了难题，这也是基于多线程的软件中出现 bug 的常见根源之一。

JS 引擎在同一时刻只能执行一段代码，所以引擎无须留意那些“可能”运行的代码。代码会被放置在作业队列（ **job queue** ）中，每当一段代码准备被执行，它就会被添加到作业队列。当 JS 引擎结束当前代码的执行后，事件循环就会执行队列中的下一个作业。事件循环（ **event loop** ）是 JS 引擎的一个内部处理进程，能监视代码的执行并管理作业队列。要记住既然是一个队列，作业就会从队列中的第一个开始，依次运行到最后一个。

事件模型

当用户点击一个按钮或按下键盘上的一个键时，一个事件（ **event** ）——例如 `onclick` ——就被触发了。该事件可能会对此交互进行响应，从而将一个新的作业添加到作业队列的尾部。这就是 JS 关于异步编程的最基本形式。事件处理程序代码直到事件发生后才会被执行，此时它会拥有合适的上下文。例如：

```
let button = document.getElementById("my-btn");
button.onclick = function(event) {
    console.log("Clicked");
};
```

在此代码中， `console.log("Clicked")` 直到 `button` 被点击后才会被执行。当 `button` 被点击，赋值给 `onclick` 的函数就被添加到作业队列的尾部，并在队列前部所有任务结束之后再执行。

事件可以很好地工作于简单的交互，但将多个分离的异步调用串联在一起却会很麻烦，因为必须追踪每个事件的事件对象（例如上例中的 `button` ）。此外，你还需确保所有的事件处理程序都能在事件第一次触发之前被绑定完毕。例如，若 `button` 在 `onclick` 被绑定之前就被点击，那就不会有任何事发生。因此虽然在响应用户交互或类似的低频功能时，事件很有用，但它在面对更复杂的需求时仍然不够灵活。

回调模式

当 Node.js 被创建后，它通过普及回调函数编程模式提升了异步编程模型。回调函数模式类似于事件模型，因为异步代码也会在未来的时间点才执行。不同之处在于需要调用的函数（即回调函数）是作为参数传入的，如下所示：

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    console.log(contents);
});
console.log("Hi!");
```

此例使用了 Node.js 惯例，即错误优先（ **error-first** ）的回调函数风格。 `readFile()` 函数用于读取由第一个参数指定的磁盘文件，并在读取完毕后执行回调函数（即第二个参数）。如果存在错误，回调函数的 `err` 参数会是一个错误对象；否则 `contents` 参数就会以字符串形式包含文件内容。

使用回调函数模式， `readFile()` 会立即开始执行，并在开始读取磁盘时暂停。这意味着 `console.log("Hi!")` 会在 `readFile()` 被调用后立即进行输出，要早于 `console.log(contents)` 的打印操作。当 `readFile()` 结束操作后，它会将回调函数以及相关参数作为一个新的作业添加到作业队列的尾部。在之前的作业全部结束后，该作业才会执行。

回调函数模式要比事件模型灵活得多，因为使用回调函数串联多个调用会相对容易。例如：

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    writeFile("example.txt", function(err) {
        if (err) {
            throw err;
        }

        console.log("File was written!");
    });
});
```

在此代码中，对于 `readFile()` 的一次成功调用引出了另一个异步调用，即调用 `writeFile()` 函数。注意这两个函数都使用了检查 `err` 的同一基本模式。当 `readFile()` 执行结束后，它添加一个作业到作业队列，假设没有出现错误， `writeFile()` 会在之后被调用。接下来， `writeFile()` 也会在执行结束后向队列添加一个作业。

这种模式运作得相当好，但当嵌套过多回调函数时，你可能会迅速察觉陷入了回调地狱（ **callback hell** ），就像这样：

```
method1(function(err, result) {

    if (err) {
        throw err;
    }

    method2(function(err, result) {

        if (err) {
            throw err;
        }

        method3(function(err, result) {

            if (err) {
                throw err;
            }

            method4(function(err, result) {

                if (err) {
                    throw err;
                }

                method5(result);
            });

        });

    });

});
```

像本例一样嵌套多个方法调用会创建错综复杂的代码，会难以理解与调试。当想要实现更复杂的功能时，回调函数也会存在问题：若想让两个异步操作并行运行，并且在它们都结束后提醒你，那该怎么做？若想同时启动两个异步操作，但只采用首个结束的结果，那又该怎么做？

在这些情况下，你需要追踪多个回调函数并做清理操作， Promise 能大幅度改善这种情况。

Promise 基础

Promise 是为异步操作的结果所准备的占位符。函数可以返回一个 Promise，而不必订阅一个事件或向函数传递一个回调参数，就像这样：

```
// readFile 承诺会在将来某个时间点完成
let promise = readFile("example.txt");
```

在此代码中， `readFile()` 实际上并未立即开始读取文件，这将会在稍后发生。此函数会返回一个 Promise 对象以表示异步读取操作，因此你可以在将来再操作它。你能对结果进行操作的确切时刻，完全取决于 Promise 的生命周期是如何进行的。

Promise 的生命周期

每个 Promise 都会经历一个短暂的生命周期，初始为进行态（ **pending** state），这表示异步操作尚未结束。一个进行中的 Promise 也被认为是未处理的（ **unsettled** ）。上个例子中的 Promise 在 `readFile()` 函数返回它的时候就是处在进行态。一旦异步操作结束， Promise 就会被认为是已处理的（ **settled** ），并进入两种可能状态之一：

1. 已完成（ **fulfilled** ）： Promise 的异步操作已成功结束；
2. 已拒绝（ **rejected** ）： Promise 的异步操作未成功结束，可能是一个错误，或由其他原因导致。

内部的 `[[PromiseState]]` 属性会被设置为 `"pending"` 、 `"fulfilled"` 或 `"rejected"` ，以反映 Promise 的状态。该属性并未在 Promise 对象上被暴露出来，因此你无法以编程方式判断 Promise 到底处于哪种状态。不过你可以使用 `then()` 方法在 Promise 的状态改变时执行一些特定操作。

> 译注：相关词汇翻译汇总
>
> Promise 是相对比较新的一个概念，相关的许多词汇有一定的交叉性，并且有些在翻译为中文时可能并不太容易分辨。因此涉及 Promise 的许多资料都对相关大部分词汇不作翻译，直接使用英文原词。
>
> 译者在本章斗胆对几乎所有词汇进行了翻译，如有不妥，欢迎指出。此处是词汇翻译的汇总，以便参考：
>
> 1. pending ：进行，表示未结束的 Promise 状态。相关词汇“进行态”。
> 2. fulfilled ：已完成，表示已成功结束的 Promise 状态，可以理解为“成功完成”。相关词汇“完成”、“被完成”、“完成态”。
> 3. rejected ：已拒绝，表示已结束但失败的 Promise 状态。相关词汇“拒绝”、“被拒绝”、“拒绝态”。
> 4. resolve ：决议，表示将 Promise 推向成功态，可以理解为“决议通过”，在 Promise 概念中与“完成”是近义词。相关词汇“决议态”、“已决议”、“被决议”。
> 5. unsettled ：未处理，或者称为“未解决”，表示 Promise 尚未被完成或拒绝，与“挂起”是近义词。
> 6. settled ：已处理，或者称为“已解决”，表示 Promise 已被完成或拒绝。注意这与“已完成”或“已决议”不同，“已处理”的状态也可能是“拒绝态”（已失败）。
> 7. fulfillment handler ：完成处理函数，表示 Promise 为完成态时会被调用的函数。
> 8. rejection handler ：拒绝处理函数，表示 Promise 为拒绝态时会被调用的函数。

`then()` 方法在所有的 Promise 上都存在，并且接受两个参数。第一个参数是 Promise 被完成时要调用的函数，与异步操作关联的任何附加数据都会被传入这个完成函数。第二个参数则是 Promise 被拒绝时要调用的函数，与完成函数相似，拒绝函数会被传入与拒绝相关联的任何附加数据。

> 用这种方式实现 `then()` 方法的任何对象都被称为一个 **thenable** 。所有的 Promise 都是 thenable ，反之则未必成立。

传递给 `then()` 的两个参数都是可选的，因此你可以监听完成与拒绝的任意组合形式。例如，研究这组 `then()` 调用：

```
let promise = readFile("example.txt");

promise.then(function(contents) {
    // 完成
    console.log(contents);
}, function(err) {
    // 拒绝
    console.error(err.message);
});

promise.then(function(contents) {
    // 完成
    console.log(contents);
});

promise.then(null, function(err) {
    // 拒绝
    console.error(err.message);
});
```

这三个 `then()` 调用都操作在同一个 Promise 上。第一个调用同时监听了完成与失败；第二个调用只监听了完成，错误不会被报告；第三个则只监听了拒绝，并不报告成功信息。

Promis 也具有一个 `catch()` 方法，其行为等同于只传递拒绝处理函数给 `then()` 。例如，以下的 `catch()` 与 `then()` 调用是功能等效的。

```
promise.catch(function(err) {
    // 拒绝
    console.error(err.message);
});

// 等同于：

promise.then(null, function(err) {
    // 拒绝
    console.error(err.message);
});
```

`then()` 与 `catch()` 背后的意图是让你组合使用它们来正确处理异步操作的结果。这个体系让操作是成功还是失败变得完全清晰，要优于事件与回调函数，事件模式倾向于在出错时不被触发，而在回调函数模式中你必须始终记得检查错误参数。关于 Promise 需要牢记的只有：若你未提供拒绝处理函数，所有的错误就会静默发生。建议始终附加一个拒绝处理函数，即使该处理程序只是用于打印错误日志。

即使完成或拒绝处理函数在 Promise 已经被处理之后才添加到作业队列，它们仍然会被执行。这允许你随时添加新的完成或拒绝处理函数，并保证它们会被调用。例如：

```
let promise = readFile("example.txt");

// 原始的完成处理函数
promise.then(function(contents) {
    console.log(contents);

    // 现在添加另一个
    promise.then(function(contents) {
        console.log(contents);
    });
});
```

在此代码中，完成处理函数又为同一个 Promise 添加了另一个完成处理函数。这个 Promise 此刻已经完成了，因此新的处理程序就被添加到任务队列，并在就绪时（前面的作业执行完毕后）被调用。拒绝处理函数使用同样方式工作。

> 每次调用 `then()` 或 `catch()` 都会创建一个新的作业，它会在 Promise 已处理时被执行。但这些作业最终会进入一个完全为 Promise 保留的作业队列。这个独立队列的确切细节对于理解如何使用 Promise 是不重要的，你只需理解作业队列通常来说是如何工作的。

创建未处理的 Promise

新的 Promise 使用 `Promise` 构造器来创建。此构造器接受单个参数：一个被称为执行器（**executor**）的函数，包含初始化 Promise 的代码。该执行器会被传递两个名为 `resolve()` 与 `reject()` 的函数作为参数。 `resolve()` 函数在执行器成功结束时被调用，用于示意该 Promise 已经准备好被决议（**resolved**），而执行器的操作失败后 `reject()` 函数则被调用。

此处有个范例，在 Node.js 中使用了一个 Promise ，实现了本章前面的 `readFile()` 函数：

```
// Node.js 范例

let fs = require("fs");

function readFile(filename) {
    return new Promise(function(resolve, reject) {

        // 触发异步操作
        fs.readFile(filename, { encoding: "utf8" }, function(err, contents) {

            // 检查错误
            if (err) {
                reject(err);
                return;
            }

            // 读取成功
            resolve(contents);

        });
    });
}

let promise = readFile("example.txt");

// 同时监听完成与拒绝
promise.then(function(contents) {
    // 完成
    console.log(contents);
}, function(err) {
    // 拒绝
    console.error(err.message);
});
```

在此例中， Node.js 原生的 `fs.readFile()` 异步调用被包装在一个 Promise 中。执行器要么传递错误对象给 `reject()` 函数，要么传递文件内容给 `resolve()` 函数。

要记住执行器会在 `readFile()` 被调用时立即运行。当 `resolve()` 或 `reject()` 在执行器内部被调用时，一个作业被添加到作业队列中，以便处理这个 Promise 。这被称为作业调度（ **job scheduling** ），若你曾用过 `setTimeout()` 或 `setInterval()` 函数，那么应该已经熟悉这种方式。在作业调度中，你添加新作业到队列中是表示："不要立刻执行这个作业，但要在稍后执行它"。例如， `setTimeout()` 函数能让你指定一个延迟时间，延迟之后作业才会被添加到队列：

```
// 在 500 毫秒之后添加此函数到作业队列
setTimeout(function() {
    console.log("Timeout");
}, 500);

console.log("Hi!");
```

此代码安排一个作业在 500 毫秒之后被添加到作业队列。此处两个 `console.log()` 调用产生了以下输出：

```
Hi!
Timeout
```

多亏这 500 毫秒的延迟，被传递给 `setTimeout()` 的匿名函数的输出，被排在了 `console.log("Hi!")` 输出之后。

> 译注：实际上前面范例中的输出顺序与 500 毫秒的延时没有关系，而与 `setTimeout()` 的机制有关。我们可以把延时改为 0 ，依然会得到相同的结果：
>
> ```
> // 在 0 毫秒之后添加此函数到作业队列
> setTimeout(function() {
>    console.log("Timeout");
> }, 0);
>
> console.log("Hi!");
> ```
>
> 输出结果会保持不变。 `setTimeout()` 确实有延时效果，但原书的例子不当，没有完全说清其中的机制。

Promise 工作方式与之相似。 Promise 的执行器会立即执行，早于源代码中在其之后的任何代码。例如：

```
let promise = new Promise(function(resolve, reject) {
    console.log("Promise");
    resolve();
});

console.log("Hi!");
```

此代码的输出结果为：

```
Promise
Hi!
```

调用 `resolve()` 触发了一个异步操作。传递给 `then()` 与 `catch()` 的函数会异步地被执行，并且它们也被添加到了作业队列（先进队列再执行）。此处有个例子：

```
let promise = new Promise(function(resolve, reject) {
    console.log("Promise");
    resolve();
});

promise.then(function() {
    console.log("Resolved.");
});

console.log("Hi!");
```

此例的输出结果为：

```
Promise
Hi!
Resolved
```

注意：尽管对 `then()` 的调用出现在 `console.log("Hi!")` 代码行之前，它实际上稍后才会执行（与执行器中那行 `"Promise"` 不同）。这是因为完成处理函数与拒绝处理函数总是会在执行器的操作结束后被添加到作业队列的尾部。

创建已处理的 Promise

基于 Promise 执行器行为的动态本质， `Promise` 构造器就是创建未处理的 Promise 的最好方式。但若你想让一个 Promise 代表一个已知的值，那么安排一个单纯传值给 `resolve()` 函数的作业并没有意义。相反，有两种方法可使用指定值来创建已处理的 Promise 。

使用 Promise.resolve()

`Promise.resolve()` 方法接受单个参数并会返回一个处于完成态的 Promise 。这意味着没有任何作业调度会发生，并且你需要向 Promise 添加一个或更多的完成处理函数来提取这个参数值。例如：

```
let promise = Promise.resolve(42);

promise.then(function(value) {
    console.log(value);         // 42
});
```

此代码创建了一个已完成的 Promise ，因此完成处理函数就接收到 42 作为 `value` 参数。若一个拒绝处理函数被添加到此 Promise ，该拒绝处理函数将永不会被调用，因为此 Promise 绝不可能转变为拒绝态。

使用 Promise.reject()

你也可以使用 `Promise.reject()` 方法来创建一个已拒绝的 Promise 。此方法像 `Promise.resolve()` 一样工作，区别是被创建的 Promise 处于拒绝态，如下：

```
let promise = Promise.reject(42);

promise.catch(function(value) {
    console.log(value);         // 42
});
```

任何附加到这个 Promise 的拒绝处理函数都将会被调用，而完成处理函数则不会执行。

> 若你传递一个 Promise 给 `Promise.resolve()` 方法，该 Promise 会不作修改原样返回。

非 Promise 的 Thenable

`Promise.resolve()` 与 `Promise.reject()` 都能接受非 Promise 的 thenable 作为参数。当传入了非 Promise 的 thenable 时，这些方法会创建一个新的 Promise ，此 Promise 会在 `then()` 函数之后被调用。

当一个对象拥有一个能接受 `resolve` 与 `reject` 参数的 `then()` 方法，该对象就会被认为是一个非 Promise 的 thenable ，就像这样：

```
let thenable = {
    then: function(resolve, reject) {
        resolve(42);
    }
};
```

此例中的 `thenable` 对象，除了 `then()` 方法之外没有任何与 Promise 相关的特征。你可以调用 `Promise.resolve()` 来将 `thenable` 转换为一个已完成的 Promise ：

```
let thenable = {
    then: function(resolve, reject) {
        resolve(42);
    }
};

let p1 = Promise.resolve(thenable);
p1.then(function(value) {
    console.log(value);     // 42
});
```

在此例中， `Promise.resolve()` 调用了 `thenable.then()` ，确定了这个 `thenable` 的 Promise 状态：由于 `resolve(42)` 在 `thenable.then()` 方法内部被调用，这个 `thenable` 的 Promise 状态也就被设为已完成。一个名为 `p1` 的新 Promise 被创建为完成态，并从 `thenable` 中接收到了值（此处为 42 ），于是 `p1` 的完成处理函数就接收到一个值为 42 的参数。

使用 `Promise.resolve()` ，同样还能从一个 thenable 创建一个已拒绝的 Promise ：

```
let thenable = {
    then: function(resolve, reject) {
        reject(42);
    }
};

let p1 = Promise.resolve(thenable);
p1.catch(function(value) {
    console.log(value);     // 42
});
```

此例类似于上例，区别是此处的 `thenable` 被拒绝了。当 `thenable.then()` 执行时，一个处于拒绝态的新 Promise 被创建，并伴随着一个值（ 42 ）。这个值此后会被传递给 `p1` 的拒绝处理函数。

`Promise.resolve()` 与 `Promise.reject()` 用类似方式工作，让你能轻易处理非 Promise 的 thenable 。在 Promise 被引入 ES6 之前，许多库都使用了 thenable ，因此将 thenable 转换为正规 Promise 的能力就非常重要了，能对之前已存在的库提供向下兼容。当你不能确定一个对象是否是 Promise 时，将该对象传递给 `Promise.resolve()` 或 `Promise.reject()` （取决于你的预期结果）是最佳可行方案，因为传入真正的 Promise 只会被直接传递出来，并不会被修改（但请注意前面译注提到的特殊情况）。

执行器错误

如果在执行器内部抛出了错误，那么 Promise 的拒绝处理函数就会被调用。例如：

```
let promise = new Promise(function(resolve, reject) {
    throw new Error("Explosion!");
});

promise.catch(function(error) {
    console.log(error.message);     // "Explosion!"
});
```

在此代码中，执行器故意抛出了一个错误。此处在每个执行器之中存在隐式的 `try-catch` ，因此错误就被捕捉并传递给了拒绝处理函数。这个例子等价于：

```
let promise = new Promise(function(resolve, reject) {
    try {
        throw new Error("Explosion!");
    } catch (ex) {
        reject(ex);
    }
});

promise.catch(function(error) {
    console.log(error.message);     // "Explosion!"
});
```

执行器处理程序捕捉了抛出的任何错误，以简化这种常见处理。但在执行器内抛出的错误仅当存在拒绝处理函数时才会被报告，否则这个错误就会被隐瞒。这在开发者早期使用 Promise 的时候是一个问题，不过 JS 环境通过提供钩子（ hook ）来捕捉被拒绝的 Promise ，从而解决了此问题。

全局的 Promise 拒绝处理

Promise 最有争议的方面之一就是：当一个 Promise 被拒绝时若缺少拒绝处理函数，就会静默失败。有人认为这是规范中最大的缺陷，因为这是 JS 语言所有组成部分中唯一未使错误清晰可见的。

由于 Promise 的本质，并不能直观判断一个 Promise 的拒绝是否已被处理。例如，研究以下示例：

```
let rejected = Promise.reject(42);

// 在此刻 rejected 不会被处理

// 一段时间后……
rejected.catch(function(value) {
    // 现在 rejected 已经被处理了
    console.log(value);
});
```

无论 Promise 是否已被解决，你都可以在任何时候调用 `then()` 或 `catch()` 并使它们正确工作，这导致很难准确知道一个 Promise 何时会被处理。此例中的 Promise 被立刻拒绝，但它后来才被处理。

虽然下个版本的 ES 可能会处理此问题，不过浏览器与 Node.js 已经实施了变更来解决开发者的这个痛点。这些变更不是 ES6 规范的一部分，但却是使用 Promise 时的宝贵工具。

Node.js 的拒绝处理

在 Node.js 中， `process` 对象上存在两个关联到 Promise 的拒绝处理的事件：

- `unhandledRejection` ：当一个 Promise 被拒绝、而在事件循环的一个轮次中没有任何拒绝处理函数被调用，该事件就会被触发；
- `rejectionHandled` ：若一个 Promise 被拒绝、并在事件循环的一个轮次之后再有拒绝处理函数被调用，该事件就会被触发。

这两个事件旨在共同帮助识别已被拒绝但未曾被处理 promise。

`unhandledRejection` 事件处理函数接受的参数是拒绝原因（常常是一个错误对象）以及已被拒绝的 Promise 。以下代码展示了 `unhandledRejection` 的应用：

```
let rejected;

process.on("unhandledRejection", function(reason, promise) {
    console.log(reason.message);            // "Explosion!"
    console.log(rejected === promise);      // true
});

rejected = Promise.reject(new Error("Explosion!"));
```

此例创建了一个带有错误对象的已被拒绝的 Promise ，并监听了 `unhandledRejection` 事件。事件处理函数接收了该错误对象作为第一个参数，原 Promise 则是第二个参数。

`rejectionHandled` 事件处理函数则只有一个参数，即已被拒绝的 Promise 。例如：

```
let rejected;

process.on("rejectionHandled", function(promise) {
    console.log(rejected === promise);              // true
});

rejected = Promise.reject(new Error("Explosion!"));

// 延迟添加拒绝处理函数
setTimeout(function() {
    rejected.catch(function(value) {
        console.log(value.message);     // "Explosion!"
    });
}, 1000);
```

此处的 `rejectionHandled` 事件在拒绝处理函数最终被调用时触发。若在 `rejected` 被创建后直接将拒绝处理函数附加到它上面，那么此事件就不会被触发。因为立即附加的拒绝处理函数在 `rejected` 被创建的事件循环的同一个轮次内就会被调用，这样 `rejectionHandled` 就不会起作用。

为了正确追踪潜在的未被处理的拒绝，使用 `rejectionHandled` 与 `unhandledRejection` 事件就能保持包含这些 Promise 的一个列表，之后等待一段时间再检查此列表。例如：

```
let possiblyUnhandledRejections = new Map();

// 当一个拒绝未被处理，将其添加到 map
process.on("unhandledRejection", function(reason, promise) {
    possiblyUnhandledRejections.set(promise, reason);
});

process.on("rejectionHandled", function(promise) {
    possiblyUnhandledRejections.delete(promise);
});

setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // 做点事来处理这些拒绝
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

这只是针对未处理的拒绝的简单追踪器。它使用了一个 Map 来储存 Promise 及其拒绝原因，每个 Promise 都是键，而它的拒绝原因就是相关的值。每当 `unhandledRejection` 被触发，Promise 及其拒绝原因就会被添加到此 Map 中。而每当 `rejectionHandled` 被触发，已被处理的 Promise 就会从这个 Map 中被移除。这样一来， `possiblyUnhandledRejections` 就会随着事件的调用而扩展或收缩。 `setInterval()` 的调用会定期检查这个列表，查看可能未被处理的拒绝，并将其信息输出到控制台（在现实情况下，你可能会想做点别的事情，以便记录或处理该拒绝）。此例使用了一个 Map 而不是 Weak Map ，这是因为你需要定期检查此 Map 来查看哪些 Promise 存在，而这是使用 Weak Map 所无法做到的。

尽管此例仅针对 Node.js ，但浏览器也实现了类似的机制来将未处理的拒绝通知给开发者。

浏览器的拒绝处理

浏览器同样能触发两个事件，来帮助识别未处理的拒绝。这两个事件会被 `window` 对象触发，并完全等效于 Node.js 的相关事件：

- `unhandledrejection` ：当一个 Promise 被拒绝、而在事件循环的一个轮次中没有任何拒绝处理函数被调用，该事件就会被触发；
- `rejectionHandled` ：若一个 Promise 被拒绝、并在事件循环的一个轮次之后再有拒绝处理函数被调用，该事件就会被触发。

Node.js 的实现会传递分离的参数给事件处理函数，而浏览器事件的处理函数则只会接收到包含下列属性的一个对象：

- `type` ： 事件的名称（ `"unhandledrejection"` 或 `"rejectionhandled"` ）；
- `promise` ：被拒绝的 Promise 对象；
- `reason` ： Promise 中的拒绝值（拒绝原因）。

浏览器的实现中存在的另一个差异就是：拒绝值（ `reason` ）在两种事件中都可用。例如：

```
let rejected;

window.onunhandledrejection = function(event) {
    console.log(event.type);                   // "unhandledrejection"
    console.log(event.reason.message);         // "Explosion!"
    console.log(rejected === event.promise);   // true
};

window.onrejectionhandled = function(event) {
    console.log(event.type);                   // "rejectionhandled"
    console.log(event.reason.message);         // "Explosion!"
    console.log(rejected === event.promise);   // true
};

rejected = Promise.reject(new Error("Explosion!"));
```

此代码使用了 DOM 0 级写法的 `onunhandledrejection` 与 `onrejectionhandled` ，对两个事件处理函数都进行了赋值（若你喜欢，也可以使用 `addEventListener("unhandledrejection")` 与 `addEventListener("rejectionhandled")` ）。每个事件处理函数都接收一个事件对象，其中包含与被拒绝的 Promise 有关的信息， `type` 、 `promise` 与 `reason` 属性都可用。

以下代码在浏览器中追踪未被处理的拒绝，与 Node.js 的代码非常相似：

```
let possiblyUnhandledRejections = new Map();

// 当一个拒绝未被处理，将其添加到 map
window.onunhandledrejection = function(event) {
    possiblyUnhandledRejections.set(event.promise, event.reason);
};

window.onrejectionhandled = function(event) {
    possiblyUnhandledRejections.delete(event.promise);
};

setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // 做点事来处理这些拒绝
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

这个实现与 Node.js 的实现几乎一模一样。使用了相同方法在 Map 中存储 Promise 及其拒绝值，并在此后进行检查。唯一真正的区别就是在事件处理函数中信息是从何处被提取出来的。

处理 Promise 的拒绝可能很麻烦，但你才刚开始见识 Promise 实际上到底有多强大。现在是时候更进一步了，把几个 Promise 串联在一起用用看。

串联 Promise

到此为止， Promise 貌似不过是在组合使用回调函数与 `setTimeout()` 函数，并进行了增量改进，然而 Promise 的内容远比表面上所看到的更多。更确切地说，存在多种方式来将 Promise 串联在一起，以完成更复杂的异步行为。

每次对 `then()` 或 `catch()` 的调用实际上创建并返回了另一个 Promise ，仅当前一个 Promise 被完成或拒绝时，后一个 Promise 才会被处理。研究以下例子：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);
}).then(function() {
    console.log("Finished");
});
```

此代码输出：

```
42
Finished
```

对 `p1.then()` 的调用返回了第二个 Promise ，又在这之上调用了 `then()` 。仅当第一个 Promise 已被决议后，第二个 `then()` 的完成处理函数才会被调用。假若你在此例中不使用串联，它看起来就会是这样：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = p1.then(function(value) {
    console.log(value);
})

p2.then(function() {
    console.log("Finished");
});
```

在这个无串联版本的代码中， `p1.then()` 的结果被存储在 `p2` 中，并且随后 `p2.then()` 被调用，以添加最终的完成处理函数。正如你可能已经猜到的，对于 `p2.then()` 的调用也返回了一个 Promise ，只是本例未使用此 Promise 。

捕获错误

Promise 链允许你捕获前一个 Promise 的完成或拒绝处理函数中发生的错误。例如：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    throw new Error("Boom!");
}).catch(function(error) {
    console.log(error.message);     // "Boom!"
});
```

在此代码中， `p1` 的完成处理函数抛出了一个错误，链式调用指向了第二个 Promise 上的 `catch()` 方法，能通过此拒绝处理函数接收前面的错误。若是一个拒绝处理函数抛出了错误，情况也是一样：

```
let p1 = new Promise(function(resolve, reject) {
    throw new Error("Explosion!");
});

p1.catch(function(error) {
    console.log(error.message);     // "Explosion!"
    throw new Error("Boom!");
}).catch(function(error) {
    console.log(error.message);     // "Boom!"
});
```

此处的执行器抛出了一个错误，就触发了 `p1` 这个 Promise 的拒绝处理函数，该处理函数随后抛出了另一个错误，并被第二个 Promise 的拒绝处理函数所捕获。链式 Promise 调用能察觉到链中其他 Promise 中的错误。

> 为了确保能正确处理任意可能发生的错误，应当始终在 Promise 链尾部添加拒绝处理函数。

在 Promise 链中返回值

Promise 链的另一重要特性是能从一个 Promise 传递数据给下一个 Promise 。前面已看到，传递给执行器中的 `resolve()` 处理函数的参数，会被传递给对应 Promise 的完成处理函数。你可以指定完成处理函数的返回值，以便沿着一个链继续传递数据。例如：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);         // "42"
    return value + 1;
}).then(function(value) {
    console.log(value);         // "43"
});
```

`p1` 的完成处理函数在被执行时返回了 `value + 1` 。由于 `value` 的值为 42 （来自执行器），此完成处理函数就返回了 43 。这个值随后被传递给第二个 Promise 的完成处理函数，并被其输出到控制台。

你能对拒绝处理函数做相同的事。当一个拒绝处理函数被调用时，它也能返回一个值。如果这么做，该值会被用于完成下一个 Promise ，就像这样：

```
let p1 = new Promise(function(resolve, reject) {
    reject(42);
});

p1.catch(function(value) {
    // 第一个完成处理函数
    console.log(value);         // "42"
    return value + 1;
}).then(function(value) {
    // 第二个完成处理函数
    console.log(value);         // "43"
});
```

此处的执行器使用 42 调用了 `reject()` ，该值被传递到这个 Promise 的拒绝处理函数中，从中又返回了 `value + 1` 。尽管后一个返回值是来自拒绝处理函数，它仍然被用于链中下一个 Promise 的完成处理函数。若有必要，一个 Promise 的失败可以通过传递返回值来恢复整个 Promise 链。

在 Promise 链中返回 Promise

从完成或拒绝处理函数中返回一个基本类型值，能够在 Promise 之间传递数据，但若你返回的是一个对象呢？若该对象是一个 Promise ，那么需要采取一个额外步骤来决定如何处理。研究以下例子：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

p1.then(function(value) {
    // 第一个完成处理函数
    console.log(value);     // 42
    return p2;
}).then(function(value) {
    // 第二个完成处理函数
    console.log(value);     // 43
});
```

在此代码中， `p1` 安排了一个决议 42 的作业， `p1` 的完成处理函数返回了一个已处于决议态的 Promise ： `p2` 。由于 `p2` 已被完成，第二个完成处理函数就被调用了。而若 `p2` 被拒绝，会调用可能存在的拒绝处理函数，而不调用第二个完成处理函数。

关于此模式需认识的首要重点是第二个完成处理函数并未被添加到 `p2` 上，而是被添加到第三个 Promise 。正因如此，上个例子就等价于：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = p1.then(function(value) {
    // 第一个完成处理函数
    console.log(value);     // 42
    return p2;
});

p3.then(function(value) {
    // 第二个完成处理函数
    console.log(value);     // 43
});
```

此处清楚说明了第二个完成处理函数被附加给 `p3` 而不是 `p2` 。这是一个细微但重要的区别，因为若 `p2` 被拒绝，则第二个完成处理函数就不会被调用。例如：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

p1.then(function(value) {
    // 第一个完成处理函数
    console.log(value);     // 42
    return p2;
}).then(function(value) {
    // 第二个完成处理函数
    console.log(value);     // 永不被调用
});
```

在此例中，由于 `p2` 被拒绝了，第二个完成处理函数就永不被调用。不过你可以改为对其附加一个拒绝处理函数：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

p1.then(function(value) {
    // 第一个完成处理函数
    console.log(value);     // 42
    return p2;
}).catch(function(value) {
    // 拒绝处理函数
    console.log(value);     // 43
});
```

此处 `p2` 被拒绝，导致拒绝处理函数被调用，来自 `p2` 的拒绝值 43 会被传递给拒绝处理函数。

从完成或拒绝处理函数中返回 thenable ，不会对 Promise 执行器何时被执行有所改变。第一个被定义的 Promise 将会首先运行它的执行器，接下来才轮到第二个 Promise 的执行器执行，以此类推。返回 thenable 只是让你能在 Promise 结果之外定义附加响应。你能通过在完成处理函数中创建一个新的 Promise ，来推迟完成处理函数的执行。例如：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);     // 42

    // 创建一个新的 promise
    let p2 = new Promise(function(resolve, reject) {
        resolve(43);
    });

    return p2
}).then(function(value) {
    console.log(value);     // 43
});
```

在此例中，一个新的 Promise 在 `p1` 的完成处理函数中被创建。这意味着直到 `p2` 被完成之后，第二个完成处理函数才会执行。若你想等待前面的 Promise 被解决，之后才去触发另一个 Promise ，那么这种模式就非常有用。

响应多个 Promise

本章至今的每个例子在同一时刻都只响应一个 Promise 。然而有时你会想监视多个 Promise 的进程，以便决定下一步行动。 ES6 提供了能监视多个 Promise 的两个方法： `Promise.all()` 与 `Promise.race()` 。

Promise.all() 方法

`Promise.all()` 方法接收单个可迭代对象（如数组）作为参数，并返回一个 Promise 。这个可迭代对象的元素都是 Promise ，只有在它们都完成后，所返回的 Promise 才会被完成。例如：

> 译注：原文在此处有貌似重复的描述，相似的话语用 被决议（ resolved ） 、 被完成（ fulfilled ）这两个术语说了两次，而这两个词在 Promise 中基本是同一个意思，因此译文删掉了其中一句。

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);

p4.then(function(value) {
    console.log(Array.isArray(value));  // true
    console.log(value[0]);              // 42
    console.log(value[1]);              // 43
    console.log(value[2]);              // 44
});
```

此处前面的每个 Promise 都用一个数值进行了决议，对 `Promise.all()` 的调用创建了新的 Promise `p4` ，在 `p1` 、 `p2` 与 `p3` 都被完成后， `p4` 最终会也被完成。传递给 `p4` 的完成处理函数的结果是一个包含每个决议值（ 42 、 43 与 44 ）的数组，这些值的存储顺序保持了待决议的 Promise 的顺序（与完成的先后顺序无关），因此你可以将结果匹配到每个 Promise 。

若传递给 `Promise.all()` 的任意 Promise 被拒绝了，那么方法所返回的 Promise 就会立刻被拒绝，而不必等待其他的 Promise 结束：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);

p4.catch(function(value) {
    console.log(Array.isArray(value))   // false
    console.log(value);                 // 43
});
```

在此例中， `p2` 被使用数值 43 进行了拒绝，则 `p4` 的拒绝处理函数就立刻被调用，而不会等待 `p1` 或 `p3` 结束执行（它们仍然会各自结束执行，只是 `p4` 不等它们）。

拒绝处理函数总会接收到单个值，而不是一个数组，该值就是被拒绝的 Promise 所返回的拒绝值。本例中的拒绝处理函数被传入了 43 ，反映了来自 `p2` 的拒绝。

Promise.race() 方法

`Promise.race()` 提供了监视多个 Promise 的一个稍微不同的方法。此方法也接受一个包含需监视的 Promise 的可迭代对象，并返回一个新的 Promise ，但一旦来源 Promise 中有一个被解决，所返回的 Promise 就会立刻被解决。与等待所有 Promise 完成的 `Promise.all()` 方法不同，在来源 Promise 中任意一个被完成时， `Promise.race()` 方法所返回的 Promise 就能作出响应。例如：

```
let p1 = Promise.resolve(42);

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.race([p1, p2, p3]);

p4.then(function(value) {
    console.log(value);     // 42
});
```

在此代码中， `p1` 被创建为一个已完成的 Promise ，而其他的 Promise 则需要调度作业。 `p4` 的完成处理函数被使用数值 42 进行了调用，并忽略了其他的 Promise 。传递给 `Promise.race()` 的 Promise 确实在进行赛跑，看哪一个首先被解决。若胜出的 Promise 是被完成，则返回的新 Promise 也会被完成；而胜出的 Promise 若是被拒绝，则新 Promise 也会被拒绝。此处有个使用拒绝的范例：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = Promise.reject(43);

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.race([p1, p2, p3]);

p4.catch(function(value) {
    console.log(value);     // 43
});
```

此处的 `p4` 被拒绝了，因为 `p2` 在 `Promise.race()` 被调用时已经处于拒绝态。尽管 `p1` 与 `p3` 都被完成，其结果仍然被忽略，因为这发生在 `p2` 被拒绝之后。

> 译注：此处范例有误。
>
> 在各个浏览器中的测试结果都是没有任何输出；而若为 `p4` 添加一个类似的完成处理函数，则会输出 `42` 。这表示在赛跑中胜出的是 `p1` 而不是 `p2` 。
>
> 如果要让此范例正确，应当在 `p1` 与 `p3` 内部的 `resolve()` 上添加延时处理。例如：
>
> ```
> let p1 = new Promise(function(resolve, reject){
>     setTimeout(function(){
>         resolve(42);
>     }, 0);
> });
> ```
>
> 由于在源代码中， `p3` 位于 `p2` 之后，因此仅修改 `p1` 也可以。

继承 Promise

正像其他内置类型，你可将一个 Promise 用作派生类的基类。这允许你自定义变异的 Promise ，在内置 Promise 的基础上扩展功能。例如，假设你想创建一个可以使用 `success()` 与 `failure()` 方法的 Promise ，对常规的 `then()` 与 `catch()` 方法进行扩展，可以像下面这样创建该 Promise 类型：

```
class MyPromise extends Promise {

    // 使用默认构造器

    success(resolve, reject) {
        return this.then(resolve, reject);
    }

    failure(reject) {
        return this.catch(reject);
    }

}

let promise = new MyPromise(function(resolve, reject) {
    resolve(42);
});

promise.success(function(value) {
    console.log(value);             // 42
}).failure(function(value) {
    console.log(value);
});
```

在此例中， `MyPromise` 从 `Promise` 上派生出来，并拥有两个附加方法。 `success()` 方法模拟了 `resolve()` ， `failure()` 方法则模拟了 `reject()` 。

每个附加方法都使用了 `this` 来调用它所模拟的方法。派生的 Promise 函数与内置的 Promise 几乎一样，除了多出来的 `success()` 与 `failure()` 方法。

由于静态方法被继承了， `MyPromise.resolve()` 方法、 `MyPromise.reject()` 方法、 `MyPromise.race()` 方法与 `MyPromise.all()` 方法在派生的 Promise 上都可用。后两个方法的行为等同于内置的方法，但前两个方法则有轻微的不同。

`MyPromise.resolve()` 与 `MyPromise.reject()` 都会返回 `MyPromise` 的一个实例，无视传递进来的值的类型，这是由于这两个方法使用了 `Symbol.species` 属性（详见第九章）来决定需要返回的 Promise 的类型。若传递内置 Promise 给这两个方法，将会被决议或被拒绝，并且会返回一个新的 `MyPromise` ，以便绑定完成或拒绝处理函数。例如：

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = MyPromise.resolve(p1);
p2.success(function(value) {
    console.log(value);         // 42
});

console.log(p2 instanceof MyPromise);   // true
```

此处的 `p1` 是一个内置的 Promise ，被传递给了 `MyPromise.resolve()` 方法。作为结果的 `p2` 是 `MyPromise` 的一个实例，来自 `p1` 的决议值被传递给了 `p2` 的完成处理函数。

若 `MyPromise` 的一个实例被传递给了 `MyPromise.resolve()` 或 `MyPromise.reject()` 方法，它会在未被解决的情况下就被直接返回。在其他情况下，这两个方法的行为都会等同于 `Promise.resolve()` 与 `Promise.reject()` 。

异步任务运行

在第八章中，我介绍了生成器，并向你展示了如何使用它来运行异步任务，就像这样：

```
let fs = require("fs");

function run(taskDef) {

    // 创建迭代器，让它在别处可用
    let task = taskDef();

    // 开始任务
    let result = task.next();

    // 递归使用函数来保持对 next() 的调用
    function step() {

        // 如果还有更多要做的
        if (!result.done) {
            if (typeof result.value === "function") {
                result.value(function(err, data) {
                    if (err) {
                        result = task.throw(err);
                        return;
                    }

                    result = task.next(data);
                    step();
                });
            } else {
                result = task.next(result.value);
```

```
                step();
            }

        }
    }

    // 开始处理过程
    step();

}

// 定义一个函数来配合任务运行器使用

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}

// 运行一个任务

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

此实现存在一些痛点。首先，将每个函数包裹在另一个函数内、再返回一个新函数，这可能会把人搞晕，实际上这句话本身就已经够混乱了。其次，返回值为函数的情况下，没有任何方法可以区分它是否应当被作为任务运行器的回调函数。

借助 Promise ，你可以确保每个异步操作都返回一个 Promise ，从而大幅度简化以及规范化异步处理，通用接口也意味着你可以大大减少异步代码。此处有一个简化任务运行器的方式：

```
let fs = require("fs");

function run(taskDef) {

    // 创建迭代器
    let task = taskDef();

    // 启动任务
    let result = task.next();

    // 递归使用函数来进行迭代
    (function step() {

        // 如果还有更多要做的
        if (!result.done) {

            // 决议一个 Promise ，让任务处理变简单
            let promise = Promise.resolve(result.value);
            promise.then(function(value) {
                result = task.next(value);
                step();
            }).catch(function(error) {
                result = task.throw(error);
                step();
            });
        }
    }());
}

// 定义一个函数来配合任务运行器使用

function readFile(filename) {
    return new Promise(function(resolve, reject) {
        fs.readFile(filename, function(err, contents) {
            if (err) {
                reject(err);
            } else {
                resolve(contents);
            }
        });
    });
}

// 运行一个任务

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

在此版本的代码中，一个通用的 `run()` 函数执行了生成器来创建一个迭代器。它调用了 `task.next()` 来启动任务，并递归调用 `step()` 直到迭代完成。

在 `step()` 函数内部，如果还有更多工作要做，那么 `result.done` 的值会是 `false` ，此时 `result.value` 应当是一个 Promise ，调用 `Promise.resolve()` 只为预防函数未正确返回 Promise （记住： `Promise.resolve()` 在被传入任意 Promise 时只会直接将其传递回来，而不是 Promise 的参数则会被包装为 Promise ）。接下来，一个完成处理函数被添加以便提取该 Promise 值，并将该值传回迭代器。此后，在 `step()` 函数调用自身之前， `result` 被赋值为下一个 yield 的结果。

一个拒绝处理函数将任意拒绝结果存储在一个错误对象中。 `task.throw()` 方法将这个错误对象传回给迭代器，而若一个错误在任务中被捕获， `result` 也会被赋值为下一个 yield 的结果，这样 `step()` 也会在 `catch()` 内部被调用，以便继续任务执行。

`run()` 函数能运行任意使用 `yield` 来实现异步代码的生成器，而不会将 Promise （或回调函数）暴露给开发者。事实上，由于函数调用后的返回值总是会被转换为一个 Promise ，该函数甚至允许返回 Promise 之外的类型。这意味着同步与异步方法在使用 `yield` 时都会正常工作，并且你永不需要检查返回值是否为一个 Promise 。

唯一需要担心的是，要确保诸如 `readFile()` 的异步方法能返回一个正确标记其状态的 Promise 。对于 Node.js 内置的方法来说，这意味着你必须转换这些方法，让它们返回 Promise 而不是使用回调函数。

> 未来的异步任务运行
>
> 在我写这本书的时候，为 JS 中的异步任务运行引入简单语法的一项工作正在进行。此工作开展在 `await` 语法上，极度借鉴了上述以 Promise 为基础的例子。其基本理念是使用一个被 `async` 标记的函数（而非生成器），并在调用另一个函数时使用 `await` 而非 `yield` ，就像这样：
>
> ```
> (async function() {
>   let contents = await readFile("config.json");
>   doSomethingWith(contents);
>   console.log("Done");
> });
> ```
>
> 在 `function` 之前的 `async` 关键字标明了此函数使用异步方式运行。 `await` 关键字则表示对于 `readFile("config.json")` 的函数调用应返回一个 Promise ，若返回类型不对，则会将其包装为 Promise 。与上述 `run()` 的实现一致， `await` 会在 Promise 被拒绝的情况下抛出错误，否则它将返回该 Promise 被决议的值。最终结果是你可以将异步代码当作同步代码来书写，仅需管理基于迭代器的状态机，不用付出额外开销。
>
> `await` 语法预计将在 ES2017 （即 ES8 ）中被最终敲定。（译注：已随着 ES2017 标准而正式启用）

总结

Promise 被设计用于改善 JS 中的异步编程，与事件及回调函数对比，在异步操作方面为你提供了更多的控制权与组合性。 Promise 调度被添加到 JS 引擎作业队列，以便稍后执行。不过此处有另一个作业队列追踪着 Promise 的完成与拒绝处理函数，以确保准确执行。

Promise 具有三种状态：进行中、已完成、已拒绝。一个 Promise 起始于进行态，并在成功时转为完成态，或在失败时转为拒绝态。在这两种情况下，处理函数都能被添加以便在 Promise 被解决后作出响应。 `then()` 方法允许你绑定完成处理函数与拒绝处理函数，而 `catch()` 方法则只允许你绑定拒绝处理函数。

你能用多种方式将多个 Promise 串联在一起，并在它们之间传递信息。每个对 `then()` 的调用都创建并返回了一个新的 Promise ，在前一个 Promise 被决议时，新 Promise 也会被决议。 Promise 链可被用于触发对一系列异步事件的响应。你还能使用 `Promise.race()` 与 `Promise.all()` 来监视多个 Promise 的进程，并进行相应的响应。

组合使用生成器与 Promise 会让异步任务运行得更容易，这是由于 Promise 提供了异步操作可返回的一个通用接口。这样你就能使用生成器与 `yield` 运算符来等待异步响应，并作出适当的应答。

多数新的 web API 都基于 Promise 创建，并且你可以期待未来会有更多的效仿之作。

第十二章 代理与反射接口

ES5 与 ES6 都推进了 JS 功能的公开。例如， JS 运行环境包含一些不可枚举、不可写入的对象属性，然而在 ES5 之前开发者无法定义他们自己的不可枚举属性或不可写入属性。 ES5 引入了 `Object.defineProperty()` 方法以便开发者在这方面能够像 JS 引擎那样做。

ES6 让开发者能进一步接近 JS 引擎的能力，这些能力原先只存在于内置对象上。语言通过代理（ **proxy** ）暴露了在对象上的内部工作，代理是一种封装，能够拦截并改变 JS 引擎的底层操作。本章会首先介绍代理所想要处理的问题，并且会讨论如何更有效地创建并使用代理。

数组的问题

在 ES6 之前， JS 的数组对象拥有特定的行为方式，开发者在自定义对象中无法对其进行模拟。当你给数组元素赋值时，数组的 `length` 属性会受到影响，同时你也可以通过修改 `length` 属性来变更数组的元素。例如：

```
let colors = ["red", "green", "blue"];

console.log(colors.length);         // 3

colors[3] = "black";

console.log(colors.length);         // 4
console.log(colors[3]);             // "black"

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
```

`colors` 开始时有三个元素。把 `"black"` 赋值给 `colors[3]` 会自动将 `length` 增加到 `4` ；而此后设置 `length` 为 `2` 则会移除数组的最末两个元素，从而只保留起始处的两个元素。在 ES5 中开发者无法模拟实现这种行为，但代理的出现改变了这种情况。

> 这种不规范行为就是 ES6 将数组认定为奇异对象的原因。

代理与反射是什么？

通过调用 `new Proxy()` ，你可以创建一个代理，用于替代另一个对象（被称为目标），这个代理对目标对象进行了虚拟，因此该代理与该目标对象表面上可以被当作同一个对象来对待。

代理允许你拦截在目标对象上的底层操作，而这原本是 JS 引擎的内部能力。拦截行为使用了一个能够响应特定操作的函数（被称为陷阱）。

反射接口由 `Reflect` 对象所代表，是给底层操作提供默认行为的方法的集合，这些操作是能够被代理重写的。每个代理陷阱都有一个对应的反射方法，每个方法都与对应的陷阱函数同名，并且接收的参数也与之一致。下表总结了这些行为：

代理陷阱	被重写的行为	默认行
`get`	读取一个属性的值	`Reflect.get()`
`set`	写入一个属性	`Reflect.set()`
`has`	`in` 运算符	`Reflect.has()`
`deleteProperty`	`delete` 运算符	`Reflect.deleteProper`
`getPrototypeOf`	`Object.getPrototypeOf()`	`Reflect.getPrototype`
`setPrototypeOf`	`Object.setPrototypeOf()`	`Reflect.setPrototype`
`isExtensible`	`Object.isExtensible()`	`Reflect.isExtensible`
`preventExtensions`	`Object.preventExtensions()`	`Reflect.preventExter`
`getOwnPropertyDescriptor`	`Object.getOwnPropertyDescriptor()`	`Reflect.getOwnProper`
`defineProperty`	`Object.defineProperty()`	`Reflect.defineProper`
`ownKeys`	`Object.keys` 、 `Object.getOwnPropertyNames()` 与 `Object.getOwnPropertySymbols()`	`Reflect.ownKeys()`
`apply`	调用一个函数	`Reflect.apply()`
`construct`	使用 `new` 调用一个函数	`Reflect.construct()`

每个陷阱函数都可以重写 JS 对象的一个特定内置行为，允许你拦截并修改它。如果你仍然需要使用原先的内置行为，则可使用对应的反射接口方法。一旦创建了代理，你就能清晰了解代理与反射接口之间的关系，因此我们最好通过一些例子来进行深入研究。

> ES6 的原始草案还有一个名为 `enumerate` 的陷阱函数，其设计意图是更改 `for-in` 与 `Object.keys()` 在对象上进行属性枚举的机制。然而，该陷阱函数在 ECMAScript 7（也被称为 ECMAScript 2016 ）中被移除了，因为它太难于实现。 `enumerate` 陷阱函数不会再出现在任何 JS 运行环境中，也不会在本章进行介绍。

创建一个简单的代理

当你使用 `Proxy` 构造器来创建一个代理时，需要传递两个参数：目标对象以及一个处理器（handler），后者是定义了一个或多个陷阱函数的对象。如果未提供陷阱函数，代理会对所有操作采取默认行为。为了创建一个仅进行转发的代理，你需要使用不包含任何陷阱函数的处理器：

```
let target = {};

let proxy = new Proxy(target, {});

proxy.name = "proxy";
console.log(proxy.name);        // "proxy"
console.log(target.name);       // "proxy"

target.name = "target";
console.log(proxy.name);        // "target"
console.log(target.name);       // "target"
```

该例中的 `proxy` 对象将所有操作直接转发给 `target` 对象。当 `proxy.name` 属性被赋值为字符串 `"proxy"` 的时候，`target.name` 属性也同时被创建，代理对象 `proxy` 自身其实并没有存储该属性，它只是简单将值转发给 `target` 对象。同样，`proxy.name` 与 `target.name` 的属性值总是相等，因为它们都指向 `target.name` ，这就意味着：为 `target.name` 设置一个新值会在 `proxy.name` 上反映出相同的改变。当然，缺少陷阱函数的代理没什么用，那么若为其定义一个陷阱函数，又会如何？

使用 set 陷阱函数验证属性值

假设你想要创建一个对象，并要求其属性值只能是数值，这就意味着该对象的每个新增属性都要被验证，并且在属性值不为数值类型时应当抛出错误。为此你需要定义 `set` 陷阱函数来重写设置属性值时的默认行为，该陷阱函数能接受四个参数：

1. `trapTarget` ：将接收属性的对象（即代理的目标对象）；
2. `key` ：需要写入的属性的键（字符串类型或符号类型）；
3. `value` ：将被写入属性的值；
4. `receiver` ：操作发生的对象（通常是代理对象）。

`Reflect.set()` 是 `set` 陷阱函数对应的反射方法，同时也是 `set` 操作的默认行为。`Reflect.set()` 方法与 `set` 陷阱函数一样，能接受这四个参数，以便在陷阱函数内部使用。该陷阱函数需要在属性被设置完成的情况下返回 `true` ，否则应当返回 `false` ，而 `Reflect.set()` 也会基于操作是否成功而返回相应的结果。

你需要使用 `set` 陷阱函数来拦截传入的 `value` 值，以便对属性值进行验证。此处有个例子：

```
let target = {
    name: "target"
};

let proxy = new Proxy(target, {
    set(trapTarget, key, value, receiver) {

        // 忽略已有属性，避免影响它们
        if (!trapTarget.hasOwnProperty(key)) {
            if (isNaN(value)) {
                throw new TypeError("Property must be a number.");
            }
        }

        // 添加属性
        return Reflect.set(trapTarget, key, value, receiver);
    }
});

// 添加一个新属性
proxy.count = 1;
console.log(proxy.count);       // 1
console.log(target.count);      // 1

// 你可以为 name 赋一个非数值类型的值，因为该属性已经存在
proxy.name = "proxy";
console.log(proxy.name);        // "proxy"
console.log(target.name);       // "proxy"

// 抛出错误
proxy.anotherName = "proxy";
```

这段代码定义了一个代理陷阱，用于对 `target` 对象新增属性的值进行验证。当执行 `proxy.count = 1` 时， `set` 陷阱函数被调用，此时 `trapTarget` 的值等于 `target` 对象， `key` 的值是字符串 `"count"` ， `value` 的值是 `1` ，而 `receiver` 的值是 `proxy` （该参数在本例中并没有被使用）。 `target` 对象上尚不存在名为 `count` 的属性，因此代理将 `value` 参数传递给 `isNaN()` 方法进行验证；如果验证结果是 `NaN` ，表示传入的属性值不是一个数值，需要抛出错误；但由于这段代码将 `count` 参数设置为 `1` ，验证通过，代理使用一致的四个参数去调用 `Reflect.set()` 方法，从而创建了一个新的属性。

当 `proxy.name` 被赋值为字符串时，操作成功完成。这是因为 `target` 对象已经拥有一个 `name` 属性，因此验证时通过调用 `trapTarget.hasOwnProperty()` 会忽略该属性，这就确保允许在该对象的已有属性上使用非数值的属性值。

当 `proxy.anotherName` 被赋值为字符串时，抛出了一个错误。这是因为该对象上并不存在 `anotherName` 属性，因此该属性的值必须被验证，而因为提供的值不是一个数值，验证过程就会抛出错误。

`set` 代理陷阱允许你在写入属性值的时候进行拦截，而 `get` 代理陷阱则允许你在读取属性值的时候进行拦截。

使用 get 陷阱函数进行对象外形验证

JS 语言有趣但有时却令人困惑的特性之一，就是读取对象不存在的属性时并不会抛出错误，而会把 `undefined` 当作该属性的值，例如：

```
let target = {};

console.log(target.name);        // undefined
```

在多数语言中，试图读取 `target.name` 属性都会抛出错误，因为该属性并不存在；但 JS 语言却会使用 `undefined` 。如果你曾经在大型代码库上进行过工作，那么你或许明白这种行为会导致严重的问题，尤其是当属性名称存在书写错误时。使用代理进行对象外形验证，有助于将你从这个错误中拯救出来。

对象外形（ **Object Shape** ）指的是对象已有的属性与方法的集合， JS 引擎使用对象外形来进行代码优化，经常会创建一些类来表示对象。如果你能大胆假设某个对象总是拥有与起始时相同的属性与方法（可以通过 `Object.preventExtensions()` 方法、 `Object.seal()` 方法或 `Object.freeze()` 方法来达到这种效果），那么在访问不存在的属性时抛出错误就会非常有用。代理能够让对象外形验证变得轻而易举。

由于该属性验证只须在读取属性时被触发，因此只要使用 `get` 陷阱函数。该陷阱函数会在读取属性时被调用，即使该属性在对象中并不存在，它能接受三个参数：

1. `trapTarget` ：将会被读取属性的对象（即代理的目标对象）；
2. `key` ：需要读取的属性的键（字符串类型或符号类型）；
3. `receiver` ：操作发生的对象（通常是代理对象）。

这些参数借鉴了 `set` 陷阱函数的参数，只有一个明显的不同，也就是没有使用 `value` 参数，因为 `get` 陷阱函数并不需要为属性写入数据。 `Reflect.get()` 方法同样接收这三个参数，并且默认会返回属性的值。

你可以使用 `get` 陷阱函数与 `Reflect.get()` 方法在目标属性不存在时抛出错误，就像这样：

```
let proxy = new Proxy({}, {
        get(trapTarget, key, receiver) {
            if (!(key in receiver)) {
                throw new TypeError("Property " + key + " doesn't exist.");
            }

            return Reflect.get(trapTarget, key, receiver);
        }
    });

// 添加属性的功能正常
proxy.name = "proxy";
console.log(proxy.name);            // "proxy"

// 读取不存在属性会抛出错误
console.log(proxy.nme);             // 抛出错误
```

在本例中， `get` 陷阱函数拦截了属性读取操作，它使用 `in` 运算符来判断 `receiver` 对象上是否已存在对应属性。使用 `receiver` 而非 `trapTarget` 去配合 `in` ，这是因为 `receiver` 本身就是拥有一个 `has` 陷阱函数（在下一节介绍）的代理对象，在此处使用 `trapTarget` 会跳过 `has` 陷阱函数，并可能给你一个错误的结果。如果要查找的属性不存在，那么就会抛出错误；否则会执行默认的行为。

这段代码允许添加 `proxy.name` 这样的新属性以供写入，并且此后可以正常读取该属性的值。最后一行代码有一个拼写错误： `proxy.nme` 应当是 `proxy.name` ，由于 `nme` 属性并不存在，程序抛出了一个错误。

使用 has 陷阱函数隐藏属性

`in` 运算符用于判断指定对象中是否存在某个属性，如果对象的属性名与指定的字符串或符号值相匹配，那么 `in` 运算符应当返回 `true` ，无论该属性是对象自身的属性还是其原型的属性。例如：

```
let target = {
    value: 42;
}

console.log("value" in target);     // true
console.log("toString" in target);  // true
```

`value` 与 `toString` 均存在于 `object` 对象中，因此 `in` 运算符都会返回 `true` 。其中 `value` 是对象自身的属性，而 `toString` 则是从 `Object` 对象上继承而来的原型属性。代理允许你使用 `has` 陷阱函数来拦截这个操作，从而在使用 `in` 运算符时返回不同的结果。

`has` 陷阱函数会在使用 `in` 运算符的情况下被调用，并且会被传入两个参数：

1. `trapTarget` ：需要读取属性的对象（即代理的目标对象）；
2. `key` ：需要检查的属性的键（字符串类型或符号类型）。

`Reflect.has()` 方法接受与之相同的参数，并向 `in` 运算符返回默认响应结果。使用 `has` 陷阱函数以及 `Reflect.has()` 方法，允许你修改部分属性在接受 `in` 检测时的行为，但保留其他属性的默认行为。例如，若只想隐藏 `value` 属性，你可以这么做：

```
let target = {
    name: "target",
    value: 42
};

let proxy = new Proxy(target, {
    has(trapTarget, key) {

        if (key === "value") {
            return false;
        } else {
            return Reflect.has(trapTarget, key);
        }
    }
});

console.log("value" in proxy);      // false
console.log("name" in proxy);       // true
console.log("toString" in proxy);   // true
```

这里的 `proxy` 对象使用了 `has` 陷阱函数，用于检查 `key` 值是否为 `"value"` 。如果是，则返回 `false` ；否则通过调用 `Reflect.has()` 方法来返回默认的结果。这样，虽然 `value` 属性确实存在于目标对象中，但 `in` 运算符却会对该属性返回 `false` ；而其他的属性（ `name` 与 `toString` ）则会正确地返回 `true` 。

使用 deleteProperty 陷阱函数避免属性被删除

`delete` 运算符能够从指定对象上删除一个属性，在删除成功时返回 `true` ，否则返回 `false` 。如果试图用 `delete` 运算符去删除一个不可配置的属性，在严格模式下将会抛出错误；而非严格模式下只是单纯返回 `false` 。此处有个例子：

```
let target = {
    name: "target",
    value: 42
};

Object.defineProperty(target, "name", { configurable: false });

console.log("value" in target);     // true

let result1 = delete target.value;
console.log(result1);               // true

console.log("value" in target);     // false

// 注：下一行代码在严格模式下会抛出错误
let result2 = delete target.name;
console.log(result2);               // false

console.log("name" in target);      // true
```

此处使用了 `delete` 运算符删除了 `value` 属性，因此在第三行代码的 `console.log()` 调用中，使用 `in` 操作符检测该属性会得到 `false` 。 `name` 属性是不可配置的，因此对其使用 `delete` 操作符只会返回 `false` 而不能删除该属性（如果代码运行在严格模式下，则会抛出错误）。你可以在代理对象中使用 `deleteProperty` 陷阱函数以改变这种行为。

`deleteProperty` 陷阱函数会在使用 `delete` 运算符去删除对象属性时被调用，并且会被传入两个参数：

1. `trapTarget` ：需要删除属性的对象（即代理的目标对象）；
2. `key` ：需要删除的属性的键（字符串类型或符号类型）。

`Reflect.deleteProperty()` 方法也接受这两个参数，并提供了 `deleteProperty` 陷阱函数的默认实现。你可以结合 `Reflect.deleteProperty()` 方法以及 `deleteProperty` 陷阱函数，来修改 `delete` 运算符的行为。例如，能确保 `value` 属性不被删除：

```
let target = {
    name: "target",
    value: 42
};

let proxy = new Proxy(target, {
    deleteProperty(trapTarget, key) {

        if (key === "value") {
            return false;
        } else {
            return Reflect.deleteProperty(trapTarget, key);
        }
    }
});

// 尝试删除 proxy.value

console.log("value" in proxy);      // true

let result1 = delete proxy.value;
console.log(result1);               // false

console.log("value" in proxy);      // true

// 尝试删除 proxy.name

console.log("name" in proxy);       // true

let result2 = delete proxy.name;
console.log(result2);               // true

console.log("name" in proxy);       // false
```

这段代码与 `has` 陷阱函数的例子相似，在 `deleteProperty` 陷阱函数中检查 `key` 的值是否为 `"value"` 。如果是，返回 `false` ；否则通过调用 `Reflect.deleteProperty()` 方法来进行默认的操作。 `value` 属性是不能被删除的，因为该操作被 `proxy` 对象拦截；而 `name` 则能如期被删除。这么做允许你在严格模式下保护属性避免其被删除，并且不会抛出错误。

原型代理的陷阱函数

第四章介绍了 `Object.setPrototypeOf()` 方法， ES6 引入该方法用于对 ES5 的 `Object.getPrototypeOf()` 方法进行补充。代理允许你通过 `setPrototypeOf` 与 `getPrototypeOf` 陷阱函数来对这两个方法的操作进行拦截。 `Object` 对象上的这两个方法都会调用代理中对应名称的陷阱函数，从而允许你改变这两个方法的行为。

由于存在着两个陷阱函数与原型代理相关联，因此分别有一组方法对应着每个陷阱函数。 `setPrototypeOf` 陷阱函数接受三个参数：

1. `trapTarget` ：需要设置原型的对象（即代理的目标对象）；
2. `proto` ：需用被用作原型的对象。

`Object.setPrototypeOf()` 方法与 `Reflect.setPrototypeOf()` 方法会被传入相同的参数。另一方面， `getPrototypeOf` 陷阱函数只接受 `trapTarget` 参数， `Object.getPrototypeOf()` 方法与 `Reflect.getPrototypeOf()` 方法也是如此。

原型代理的陷阱函数如何工作

这些陷阱函数受到一些限制。首先， `getPrototypeOf` 陷阱函数的返回值必须是一个对象或者 `null` ，其他任何类型的返回值都会引发“运行时”错误。对于返回值的检测确保了 `Object.getPrototypeOf()` 会返回预期的结果。其次， `setPrototypeOf` 必须在操作没有成功的情况下返回 `false` ，这样会让 `Object.setPrototypeOf()` 抛出错误；而若 `setPrototypeOf` 的返回值不是 `false` ，则 `Object.setPrototypeOf()` 就会认为操作已成功。

下面这个例子通过返回 `null` 隐藏了代理对象的原型，并且使得该原型不可被修改：

```
let target = {};
let proxy = new Proxy(target, {
    getPrototypeOf(trapTarget) {
        return null;
    },
    setPrototypeOf(trapTarget, proto) {
        return false;
    }
});

let targetProto = Object.getPrototypeOf(target);
let proxyProto = Object.getPrototypeOf(proxy);

console.log(targetProto === Object.prototype);      // true
console.log(proxyProto === Object.prototype);       // false
console.log(proxyProto);                            // null

// 成功
Object.setPrototypeOf(target, {});

// 抛出错误
Object.setPrototypeOf(proxy, {});
```

这段代码突出了 `target` 对象与 `proxy` 对象的行为差异。使用 `target` 对象作为参数调用 `Object.getPrototypeOf()` 会返回一个对象值；而使用 `proxy` 对象调用该方法则会返回 `null` ，因为 `getPrototypeOf` 陷阱函数被调用了。类似的，使用 `target` 去调用 `Object.setPrototypeOf()` 会成功；而由于 `setPrototypeOf` 陷阱函数的存在，使用 `proxy` 则会引发错误。

如果你想在这两个陷阱函数中使用默认的行为，那么只需调用 `Reflect` 对象上的相应方法。例如，下面的代码为 `getPrototypeOf` 方法与 `setPrototypeOf` 方法实现了默认的行为：

```
let target = {};
let proxy = new Proxy(target, {
    getPrototypeOf(trapTarget) {
        return Reflect.getPrototypeOf(trapTarget);
    },
    setPrototypeOf(trapTarget, proto) {
        return Reflect.setPrototypeOf(trapTarget, proto);
    }
});

let targetProto = Object.getPrototypeOf(target);
let proxyProto = Object.getPrototypeOf(proxy);

console.log(targetProto === Object.prototype);      // true
console.log(proxyProto === Object.prototype);       // true

// 成功
Object.setPrototypeOf(target, {});

// 同样成功
Object.setPrototypeOf(proxy, {});
```

在这个例子中，你可以将 `target` 对象与 `proxy` 对象互换使用，因为 `getPrototypeOf` 与 `setPrototypeOf` 陷阱函数只是直接传递参数去调用默认的实现。需要特别注意的是，本例使用了 `Reflect.getPrototypeOf()` 方法与 `Reflect.setPrototypeOf()` 方法，而没有使用 `Object` 对象上的同名方法，因为这些方法存在重要差别。

为何存在两组方法？

关于 `Reflect.getPrototypeOf()` 与 `Reflect.setPrototypeOf()` ，令人困惑的是它们看起来与 `Object.getPrototypeOf()` 与 `Object.setPrototypeOf()` 非常相似。然而虽然两组方法分别进行着相似的操作，它们之间仍然存在显著差异。

`Object.getPrototypeOf()` 与 `Object.setPrototypeOf()` 属于高级操作，从产生之初便已提供给开发者使用；而 `Reflect.getPrototypeOf()` 与 `Reflect.setPrototypeOf()` 属于底层操作，允许开发者访问 `[[GetPrototypeOf]]` 与 `[[SetPrototypeOf]]` 这两个原先仅供语言内部使用的操作。 `Reflect.getPrototypeOf()` 方法是对内部的 `[[GetPrototypeOf]]` 操作的封装（并附加了一些输入验证），而 `Reflect.setPrototypeOf()` 方法与 `[[SetPrototypeOf]]` 操作之间也存在类似的关系。虽然 `Object` 对象上的同名方法也调用了 `[[GetPrototypeOf]]` 与 `[[SetPrototypeOf]]` ，但它们在调用这两个操作之前添加了一些步骤，并检查返回值以决定如何行动。

`Reflect.getPrototypeOf()` 方法在接收到的参数不是一个对象时会抛出错误，而 `Object.getPrototypeOf()` 则会在操作之前先将参数值转换为一个对象。如果你分别传入一个数值给这两个方法，会得到截然不同的结果：

```
let result1 = Object.getPrototypeOf(1);
console.log(result1 === Number.prototype);  // true

// 抛出错误
Reflect.getPrototypeOf(1);
```

`Object.getPrototypeOf()` 方法能够为数值 `1` 找到一个原型，因为它首先会将数值 `1` 转换为一个 `Number` 对象，这样就可以使用 `Number` 对象的原型。而 `Reflect.getPrototypeOf()` 方法并不会转换这个参数，由于数值 `1` 不是一个对象，因此该方法调用会导致一个错误。

`Reflect.setPrototypeOf()` 方法与 `Object.setPrototypeOf()` 方法也有差异。最重要的是，`Reflect.setPrototypeOf()` 方法返回一个布尔值用于表示操作是否已成功，成功时返回 `true`，而失败时返回 `false`；但若 `Object.setPrototypeOf()` 方法的操作失败，它会抛出错误。

在“原型代理的陷阱函数如何工作”那个小节的第一个例子中，当 `setPrototypeOf` 代理陷阱返回 `false` 时，它导致 `Object.setPrototypeOf()` 方法抛出了错误。此外，`Object.setPrototypeOf()` 方法会将传入的第一个参数作为自身的返回值，因此并不适合用来实现 `setPrototypeOf` 代理陷阱的默认行为。下面的代码演示了这些区别：

```
let target1 = {};
let result1 = Object.setPrototypeOf(target1, {});
console.log(result1 === target1);                   // true

let target2 = {};
let result2 = Reflect.setPrototypeOf(target2, {});
console.log(result2 === target2);                   // false
console.log(result2);                               // true
```

在本例中，`Object.setPrototypeOf()` 方法将 `target1` 对象作为返回值，而 `Reflect.setPrototypeOf()` 方法则返回了 `true`。这个微妙的差异非常重要，虽然 `Object` 对象与 `Reflect` 对象貌似存在重复的方法，但在代理陷阱内却必须使用 `Reflect` 对象上的方法。

> 在使用代理时，这两组方法都会调用 `getPrototypeOf` 与 `setPrototypeOf` 陷阱函数。

对象可扩展性的陷阱函数

ES5 通过 `Object.preventExtensions()` 与 `Object.isExtensible()` 方法给对象增加了可扩展性。而 ES6 则通过 `preventExtensions` 与 `isExtensible` 陷阱函数允许代理拦截对于底层对象的方法调用。这两个陷阱函数都接受名为 `trapTarget` 的单个参数，此参数代表方法在哪

个对象上被调用。 `isExtensible` 陷阱函数必须返回一个布尔值用于表明目标对象是否可被扩展，而 `preventExtensions` 陷阱函数也需要返回一个布尔值，用于表明操作是否已成功。

同时也存在 `Reflect.preventExtensions()` 与 `Reflect.isExtensible()` 方法，用于实现默认的行为。这两个方法都返回布尔值，因此它们可以在对应的陷阱函数内直接使用。

两个基本范例

为了弄懂对象可扩展性的陷阱函数如何运作，可研究如下代码，该代码实现了 `isExtensible` 与 `preventExtensions` 陷阱函数的默认行为。

```
let target = {};
let proxy = new Proxy(target, {
    isExtensible(trapTarget) {
        return Reflect.isExtensible(trapTarget);
    },
    preventExtensions(trapTarget) {
        return Reflect.preventExtensions(trapTarget);
    }
});

console.log(Object.isExtensible(target));       // true
console.log(Object.isExtensible(proxy));        // true

Object.preventExtensions(proxy);

console.log(Object.isExtensible(target));       // false
console.log(Object.isExtensible(proxy));        // false
```

这个例子将 `Object.preventExtensions()` 与 `Object.isExtensible()` 方法直接从 `proxy` 对象传递到 `target` 对象。当然，你也可以自行修改这种行为。例如，如果不想让代理上的 `Object.preventExtensions()` 操作成功，你可以强制 `preventExtensions` 陷阱函数返回 `false` 。

```
let target = {};
let proxy = new Proxy(target, {
    isExtensible(trapTarget) {
        return Reflect.isExtensible(trapTarget);
    },
    preventExtensions(trapTarget) {
        return false
    }
});

console.log(Object.isExtensible(target));       // true
console.log(Object.isExtensible(proxy));        // true

Object.preventExtensions(proxy);

console.log(Object.isExtensible(target));       // true
console.log(Object.isExtensible(proxy));        // true
```

这段代码中，对于 `Object.preventExtensions(proxy)` 的调用被有效地忽略了。因为 `preventExtensions` 陷阱函数返回了 `false` ，因此该操作并不会被传递到 `target` 对象上，于是后面的 `Object.isExtensible()` 仍然会返回 `true` 。

> 译注：此代码在 FireFox 和 Edge 中能够正常执行，但在 Chrome 中却会在 `Object.preventExtensions(proxy)` 这一行抛出错误。

可扩展性的重复方法

你可能已经注意到：在可扩展性方面， `Object` 对象与 `Reflect` 对象再次出现了重复的方法。不过它们之间的差异相对要小得多： `Object.isExtensible()` 方法与 `Reflect.isExtensible()` 方法几乎一样，只在接收到的参数不是一个对象时才有例外。此时 `Object.isExtensible()` 总是会返回 `false` ，而 `Reflect.isExtensible()` 则会抛出一个错误。此处有个示例：

```
let result1 = Object.isExtensible(2);
console.log(result1);                       // false

// 抛出错误
let result2 = Reflect.isExtensible(2);
```

这种区别与 `Object.getPrototypeOf()` 方法和 `Reflect.getPrototypeOf()` 方法之间的区别相似，底层功能的方法与对应的高层方法相比，会进行更严格的错误检查。

`Object.preventExtensions()` 方法与 `Reflect.preventExtensions()` 方法也是非常相似的。 `Object.preventExtensions()` 方法总是将传递给它的参数值作为自身的返回值，即使该参数不是一个对象；而另一方面 `Reflect.preventExtensions()` 方法则会在参数不是对象时抛出错

误。当参数确实是一个对象时， `Reflect.preventExtensions()` 会在操作成功时返回 `true` ，否则返回 `false` 。例如：

```
let result1 = Object.preventExtensions(2);
console.log(result1);                               // 2

let target = {};
let result2 = Reflect.preventExtensions(target);
console.log(result2);                               // true

// 抛出错误
let result3 = Reflect.preventExtensions(2);
```

此代码中的 `Object.preventExtensions()` 将传递给它的参数 `2` 作为返回值，尽管 `2` 并不是一个对象。 而 `Reflect.preventExtensions()` 方法在接收一个对象作为参数时返回了 `true` ，但在接收 `2` 时抛出了错误。

属性描述符的陷阱函数

ES5 最重要的特征之一就是引入了 `Object.defineProperty()` 方法用于定义属性的特性。在 JS 之前的版本中，没有方法可以定义一个访问器属性，也不能让属性变成只读或是不可枚举。而这些特性都能够利用 `Object.defineProperty()` 方法来实现，并且你还可以利用 `Object.getOwnPropertyDescriptor()` 方法来检索这些特性。

代理允许你使用 `defineProperty` 与 `getOwnPropertyDescriptor` 陷阱函数，来分别拦截对于 `Object.defineProperty()` 与 `Object.getOwnPropertyDescriptor()` 的调用。 `defineProperty` 陷阱函数接受下列三个参数：

1. `trapTarget` ：需要被定义属性的对象（即代理的目标对象）；
2. `key` ：属性的键（字符串类型或符号类型）；
3. `descriptor` ：为该属性准备的描述符对象。

`defineProperty` 陷阱函数要求你在操作成功时返回 `true` ，否则返回 `false` 。 `getOwnPropertyDescriptor` 陷阱函数则只接受 `trapTarget` 与 `key` 这两个参数，并会返回对应的描述符。 `Reflect.defineProperty()` 与 `Reflect.getOwnPropertyDescriptor()` 方法作为上述陷阱函数的对应方法，接受与之相同的参数。此处有个例子，实现了每个陷阱函数的默认行为：

```
let proxy = new Proxy({}, {
    defineProperty(trapTarget, key, descriptor) {
        return Reflect.defineProperty(trapTarget, key, descriptor);
    },
    getOwnPropertyDescriptor(trapTarget, key) {
        return Reflect.getOwnPropertyDescriptor(trapTarget, key);
    }
});

Object.defineProperty(proxy, "name", {
    value: "proxy"
});

console.log(proxy.name);            // "proxy"

let descriptor = Object.getOwnPropertyDescriptor(proxy, "name");

console.log(descriptor.value);      // "proxy"
```

这段代码使用了 `Object.defineProperty()` 方法在代理对象上定义了名为 `"name"` 的属性，该属性的描述符可以使用 `Object.getOwnPropertyDescriptor()` 方法进行检索。

阻止 Object.defineProperty()

`defineProperty` 陷阱函数要求你返回一个布尔值用于表示操作是否已成功。当它返回 `true` 时， `Object.defineProperty()` 会正常执行；而如果它返回了 `false` ，则 `Object.defineProperty()` 会抛出错误。 你可以使用该功能来限制哪些属性可以被 `Object.defineProperty()` 方法定义。例如，如果想阻止定义符号类型的属性，你可以检查传入的键是否为符号，若是则返回 `false` ，就像这样：

```
let proxy = new Proxy({}, {
    defineProperty(trapTarget, key, descriptor) {

        if (typeof key === "symbol") {
            return false;
        }

        return Reflect.defineProperty(trapTarget, key, descriptor);
    }
});

Object.defineProperty(proxy, "name", {
    value: "proxy"
});

console.log(proxy.name);                    // "proxy"

let nameSymbol = Symbol("name");

// 抛出错误
Object.defineProperty(proxy, nameSymbol, {
    value: "proxy"
});
```

当 `key` 是一个符号时， `defineProperty` 代理陷阱会返回 `false` ，而其他情况下则会保持默认的行为。当使用字符串 `"name"` 作为键去调用 `Object.defineProperty()` 时，该方法能够成功执行；然而当使用符号变量 `nameSymbol` 去调用 `Object.defineProperty()` 的时候， `defineProperty` 陷阱函数返回了 `false` ，导致程序抛出了错误。

> 你可以让陷阱函数返回 `true` ，同时不去调用 `Reflect.defineProperty()` 方法，这样 `Object.defineProperty()` 就会静默失败，如此便可在未实际去定义属性的情况下抑制运行错误。

描述符对象的限制

为了确保 `Object.defineProperty()` 与 `Object.getOwnPropertyDescriptor()` 方法的行为一致，传递给 `defineProperty` 陷阱函数的描述符对象必须是正规的。出于同一原因， `getOwnPropertyDescriptor` 陷阱函数返回的对象也始终需要被验证。

任意对象都能作为 `Object.defineProperty()` 方法的第三个参数；然而传递给 `defineProperty` 陷阱函数的描述符对象参数，则只有 `enumerable` 、 `configurable` 、 `value` 、 `writable` 、 `get` 与 `set` 这些属性是被许可的。例如：

```
let proxy = new Proxy({}, {
    defineProperty(trapTarget, key, descriptor) {
        console.log(descriptor.value);              // "proxy"
        console.log(descriptor.name);               // undefined

        return Reflect.defineProperty(trapTarget, key, descriptor);
    }
});


Object.defineProperty(proxy, "name", {
    value: "proxy",
    name: "custom"
});
```

此代码中调用 `Object.defineProperty()` 时，在第三个参数上使用了一个非标准的 `name` 属性。当 `defineProperty` 陷阱函数被调用时， `descriptor` 对象不会拥有 `name` 属性，只拥有一个 `value` 属性。这是因为 `descriptor` 对象实际上并不是原先传递给 `Object.defineProperty()` 方法的第三个参数，而是一个新的对象，其中只包含了被许可的属性（因此 `name` 属性被丢弃了）。 `Reflect.defineProperty()` 方法同样也会忽略描述符上的非标准属性。

`getOwnPropertyDescriptor` 陷阱函数有一个微小差异，要求返回值必须是 `null` 、 `undefined` ，或者是一个对象。如果返回值是一个对象，则只允许该对象拥有 `enumerable` 、 `configurable` 、 `value` 、 `writable` 、 `get` 或 `set` 这些自有属性。如果你返回的对象包含了不被许可的自有属性，则程序会抛出错误，就像下面演示的这样：

```
let proxy = new Proxy({}, {
    getOwnPropertyDescriptor(trapTarget, key) {
        return {
            name: "proxy"
        };
    }
});

// 抛出错误
let descriptor = Object.getOwnPropertyDescriptor(proxy, "name");
```

`name` 属性在属性描述符中是不被许可的，因此当 `Object.getOwnPropertyDescriptor()` 被调用时， `getOwnPropertyDescriptor` 的返回值会触发一个错误。这个限制保证了 `Object.getOwnPropertyDescriptor()` 的返回值总是拥有可信任的结构，无论是否使用了代理。

重复的描述符方法

ES6 再次出现了令人困惑的相似方法， `Object.defineProperty()` 和 `Object.getOwnPropertyDescriptor()` 方法貌似分别与 `Reflect.defineProperty()` 和 `Reflect.getOwnPropertyDescriptor()` 方法相同。正如本章之前讨论过的那些配套方法一样，这些方法也存在一些微小但重要的差异。

defineProperty() 方法

`Object.defineProperty()` 方法与 `Reflect.defineProperty()` 方法几乎一模一样，只是返回值有区别。前者返回调用它时的第一个参数，而后者在操作成功时返回 `true` 、失败时返回 `false` 。例如：

```
let target = {};

let result1 = Object.defineProperty(target, "name", { value: "target "});

console.log(target === result1);        // true

let result2 = Reflect.defineProperty(target, "name", { value: "reflect" });

console.log(result2);                   // true
```

使用 `target` 对象去调用 `Object.defineProperty()` 方法，返回值也是 `target` 。而同样使用 `target` 对象去调用 `Reflect.defineProperty()` ，返回值却是 `true` ，表示操作已经成功。由于 `defineProperty` 代理陷阱需要一个布尔值作为返回值，因此最好在必要时使用 `Reflect.defineProperty()` 来实现默认的行为。

getOwnPropertyDescriptor() 方法

`Object.getOwnPropertyDescriptor()` 方法会在接收的第一个参数是一个基本类型值时，将该参数转换为一个对象。另一方面， `Reflect.getOwnPropertyDescriptor()` 方法则会在第一个参数是基本类型值的时候抛出错误。下面这个例子展示了二者的特性：

```
let descriptor1 = Object.getOwnPropertyDescriptor(2, "name");
console.log(descriptor1);       // undefined

// 抛出错误
let descriptor2 = Reflect.getOwnPropertyDescriptor(2, "name");
```

此代码中的 `Object.getOwnPropertyDescriptor()` 方法返回了 `undefined` ，因为它将 `2` 转换为一个对象，转换后的对象并不包含 `name` 属性，而返回 `undefined` 是指定属性名在目标对象中不存在时的标准行为。然而当 `Reflect.getOwnPropertyDescriptor()` 被调用时，立刻抛出了一个错误，因为该方法不接受基本类型值作为它的第一个参数。

ownKeys 陷阱函数

`ownKeys` 代理陷阱拦截了内部方法 `[[OwnPropertyKeys]]` ，并允许你返回一个数组用于重写该行为。返回的这个数组会被用于四个方法： `Object.keys()` 方法、 `Object.getOwnPropertyNames()` 方法、 `Object.getOwnPropertySymbols()` 方法与 `Object.assign()` 方法，其中 `Object.assign()` 方法会使用该数组来决定哪些属性会被复制。

`ownKeys` 陷阱函数的默认行为由 `Reflect.ownKeys()` 方法实现，会返回一个由全部自有属性的键构成的数组，无论键的类型是字符串还是符号。 `Object.getOwnProperyNames()` 方法与 `Object.keys()` 方法会将符号值从该数组中过滤出去；相反， `Object.getOwnPropertySymbols()` 会将字符串值过滤掉；而 `Object.assign()` 方法会使用数组中所有的字符串值与符号值。

`ownKeys` 陷阱函数接受单个参数，即目标对象，同时必须返回一个数组或者一个类数组对象，不合要求的返回值会导致错误。你可以使用 `ownKeys` 陷阱函数去过滤特定的属性，以避免这些属性被 `Object.keys()` 方法、 `Object.getOwnPropertyNames()` 方法、 `Object.getOwnPropertySymbols()` 方法或 `Object.assign()` 方法使用。假设你不想在结果中包含任何以下划线打头的属性（在 JS 的编码惯例中，这代表该字段是私有的），那么可以使用 `ownKeys` 陷阱函数来将它们过滤掉，就像下面这样：

```
let proxy = new Proxy({}, {
    ownKeys(trapTarget) {
        return Reflect.ownKeys(trapTarget).filter(key => {
            return typeof key !== "string" || key[0] !== "_";
        });
    }
});

let nameSymbol = Symbol("name");

proxy.name = "proxy";
proxy._name = "private";
proxy[nameSymbol] = "symbol";

let names = Object.getOwnPropertyNames(proxy),
    keys = Object.keys(proxy);
    symbols = Object.getOwnPropertySymbols(proxy);

console.log(names.length);      // 1
console.log(names[0]);          // "name"

console.log(keys.length);      // 1
console.log(keys[0]);          // "name"

console.log(symbols.length);    // 1
console.log(symbols[0]);        // "Symbol(name)"
```

这个例子使用了一个 `ownKeys` 陷阱函数，首先调用了 `Reflect.ownKeys()` 方法来获取目标对象的键列表；接下来， `filter()` 方法被用于将所有下划线打头的字符串类型的键过滤出去；这之后向 `proxy` 对象添加了三个属性： `name` 、 `_name` 与 `nameSymbol` 。当 `proxy` 对象上的 `Object.getOwnPropertyNames()` 方法与 `Object.keys()` 方法被调用时，只获得了 `name` 属性；类似的， `Object.getOwnPropertySymbols()` 方法被调用时只获得了 `nameSymbol` 属性；而 `_name` 属性则始终没有出现在结果里，因为它被过滤了。

> `ownKeys` 陷阱函数也能影响 `for-in` 循环，因为这种循环调用了陷阱函数来决定哪些值能够被用在循环内。

使用 apply 与 construct 陷阱函数的函数代理

在所有的代理陷阱中，只有 `apply` 与 `construct` 要求代理目标对象必须是一个函数。回忆一下第三章的内容，函数拥有两个内部方法： `[[Call]]` 与 `[[Construct]]` ，前者会在函数被直接调用时执行，而后者会在函数被使用 `new` 运算符调用时执行。 `apply` 与 `construct` 陷阱函数对应着这两个内部方法，并允许你对其进行重写。当不使用 `new` 去调用一个函数时， `apply` 陷阱函数会接收到下列三个参数（ `Reflect.apply()` 也会接收这些参数）：

1. `trapTarget` ：被执行的函数（即代理的目标对象）；
2. `thisArg` ：调用过程中函数内部的 `this` 值；
3. `argumentsList` ：被传递给函数的参数数组。

当使用 `new` 去执行函数时， `construct` 陷阱函数会被调用并接收到下列两个参数：

1. `trapTarget` ：被执行的函数（即代理的目标对象）；
2. `argumentsList` ：被传递给函数的参数数组。

`Reflect.construct()` 方法同样会接收到这两个参数，还会收到可选的第三参数 `newTarget` ，如果提供了此参数，则它就指定了函数内部的 `new.target` 值。

`apply` 与 `construct` 陷阱函数结合起来就完全控制了任意的代理目标对象函数的行为。为了模拟函数的默认行为，你可以这么做：

```
let target = function() { return 42 },
    proxy = new Proxy(target, {
        apply: function(trapTarget, thisArg, argumentList) {
            return Reflect.apply(trapTarget, thisArg, argumentList);
        },
        construct: function(trapTarget, argumentList) {
            return Reflect.construct(trapTarget, argumentList);
        }
    });

// 使用了函数的代理，其目标对象会被视为函数
console.log(typeof proxy);              // "function"

console.log(proxy());                   // 42

var instance = new proxy();
console.log(instance instanceof proxy);     // true
console.log(instance instanceof target);    // true
```

本例中的函数会返回一个数值 `42` 。该函数的代理使用了 `apply` 与 `construct` 陷阱函数来将对应行为分别委托给 `Reflect.apply()` 与 `Reflect.construct()` 方法。最终结果是代理函数就像目标函数一样工作，包括使用 `typeof` 会将其检测为函数，并且使用 `new` 运算符调用会产生一个实例对象 `instance` 。 `instance` 对象会被同时判定为 `proxy` 与 `target` 对象的实例，是因为 `instanceof` 运算符使用了原型链来进行推断，而原型链查找并没有受到这个代理的影响，因此 `proxy` 对象与 `target` 对象对于 JS 引擎来说就有同一个原型。

验证函数的参数

`apply` 与 `construct` 陷阱函数在函数的执行方式上开启了很多的可能性。例如，假设你想要保证所有参数都是某个特定类型的，可使用 `apply` 陷阱函数来进行验证:

```
// 将所有参数相加
function sum(...values) {
    return values.reduce((previous, current) => previous + current, 0);
}

let sumProxy = new Proxy(sum, {
        apply: function(trapTarget, thisArg, argumentList) {

            argumentList.forEach((arg) => {
                if (typeof arg !== "number") {
                    throw new TypeError("All arguments must be numbers.");
                }
            });

            return Reflect.apply(trapTarget, thisArg, argumentList);
        },
        construct: function(trapTarget, argumentList) {
            throw new TypeError("This function can't be called with new.");
        }
    });

console.log(sumProxy(1, 2, 3, 4));          // 10

// 抛出错误
console.log(sumProxy(1, "2", 3, 4));

// 同样抛出错误
let result = new sumProxy();
```

此例使用了 `apply` 陷阱函数来确保所有的参数都是数值。 `sum()` 函数会将所有传递进来的参数值相加，如果传入参数的值不是数值类型，该函数仍然会尝试加法操作，这样可能会导致意外的结果。此代码通过将 `sum()` 函数封装在 `sumProxy()` 代理中，在函数运行之前拦截了函数调用，以保证每个参数都是数值。出于安全的考虑，这段代码使用 `construct` 陷阱抛出错误，以确保该函数不会被使用 `new` 运算符调用。

相反的，你也可以限制函数必须使用 `new` 运算符调用，同时确保它的参数都是数值：

```
function Numbers(...values) {
    this.values = values;
}

let NumbersProxy = new Proxy(Numbers, {

        apply: function(trapTarget, thisArg, argumentList) {
            throw new TypeError("This function must be called with new.");
        },

        construct: function(trapTarget, argumentList) {
            argumentList.forEach((arg) => {
                if (typeof arg !== "number") {
                    throw new TypeError("All arguments must be numbers.");
                }
            });

            return Reflect.construct(trapTarget, argumentList);
        }
    });

let instance = new NumbersProxy(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]

// 抛出错误
NumbersProxy(1, 2, 3, 4);
```

此代码中的 `apply` 陷阱函数会抛出错误，而 `construct` 陷阱函数则使用了 `Reflect.construct()` 方法来验证输入并返回一个新的实例。当然，你也可以不必使用代理，而是用 `new.target` 来完成相同的功能。

调用构造器而无须使用 new

第三章曾介绍了 `new.target` 元属性，在使用 `new` 运算符调用函数时，这个属性就是对该函数的一个引用。这意味着你可以使用 `new.target` 来判断函数被调用时是否使用了 `new` ，就像这样：

```
function Numbers(...values) {

    if (typeof new.target === "undefined") {
        throw new TypeError("This function must be called with new.");
    }

    this.values = values;
}

let instance = new Numbers(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]

// 抛出错误
Numbers(1, 2, 3, 4);
```

这个例子在不使用 `new` 来调用 `Numbers` 函数的情况下抛出了错误，与“验证函数的参数”那个小节的例子效果一致，但并没有使用代理。相对于使用代理，这种写法更简单，并且若只想阻止不使用 `new` 来调用函数的行为，这种写法也更胜一筹。然而有时你所要修改其行为的函数是你所无法控制的，此时使用代理就有意义了。

假设 `Numbers` 函数是硬编码的，无法被修改，已知该代码依赖于 `new.target` ，而你想要在调用函数时避免这个检查。在“必须使用 `new` ”这一限制已经确定的情况下，你可以使用 `apply` 陷阱函数来规避它：

```
function Numbers(...values) {

    if (typeof new.target === "undefined") {
        throw new TypeError("This function must be called with new.");
    }

    this.values = values;
}

let NumbersProxy = new Proxy(Numbers, {
        apply: function(trapTarget, thisArg, argumentsList) {
            return Reflect.construct(trapTarget, argumentsList);
        }
    });


let instance = NumbersProxy(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]
```

`NumbersProxy` 函数允许你调用 `Numbers` 而无须使用 `new` ，并且让这种调用的效果与使用了 `new` 的情况保持一致。为此， `apply` 陷阱函数使用传给自身的参数去对 `Reflect.construct()` 方法进行了调用，于是 `Numbers` 内部的 `new.target` 就被设置为

`Numbers` ，从而避免抛出错误。尽管这只是修改 `new.target` 的一个简单例子，但你还可以做得更加直接。

重写抽象基础类的构造器

你可以进一步指定 `Reflect.construct()` 的第三个参数，用于给 `new.target` 赋值。当函数把 `new.target` 与已知值进行比较的时候，例如在创建一个抽象基础类的构造器的场合下（参阅第九章），这么做会很有帮助。在抽象基础类的构造器中， `new.target` 被要求不能是构造器自身，正如这个例子：

```
class AbstractNumbers {

    constructor(...values) {
        if (new.target === AbstractNumbers) {
            throw new TypeError("This function must be inherited from.");
        }

        this.values = values;
    }
}

class Numbers extends AbstractNumbers {}

let instance = new Numbers(1, 2, 3, 4);
console.log(instance.values);           // [1,2,3,4]

// 抛出错误
new AbstractNumbers(1, 2, 3, 4);
```

当 `new AbstractNumbers()` 被调用时， `new.target` 等于 `AbstractNumbers` ，从而抛出了错误；而调用 `new Numbers()` 能正常工作，因为此时 `new.target` 等于 `Numbers` 。你可以使用代理手动指定 `new.target` 从而绕过这个限制：

```
class AbstractNumbers {

    constructor(...values) {
        if (new.target === AbstractNumbers) {
            throw new TypeError("This function must be inherited from.");
        }

        this.values = values;
    }
}

let AbstractNumbersProxy = new Proxy(AbstractNumbers, {
        construct: function(trapTarget, argumentList) {
            return Reflect.construct(trapTarget, argumentList, function() {});
        }
    });


let instance = new AbstractNumbersProxy(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]
```

`AbstractNumbersProxy` 使用 `construct` 陷阱函数拦截了对于 `new AbstractNumbersProxy()` 方法的调用，这样陷阱函数就将一个空函数作为第三个参数传递给了 `Reflect.construct()` 方法，让这个空函数成为构造器内部的 `new.target` 。由于此时 `new.target` 的值并不等于 `AbstractNumbers` ，就不会抛出错误，构造器可以执行完成。

可被调用的类构造器

第九章说明了构造器必须始终使用 `new` 来调用，原因是类构造器的内部方法 `[[Call]]` 被明确要求抛出错误。然而代理可以拦截对于 `[[Call]]` 方法的调用，意味着你可以借助代理有效创建一个可被调用的类构造器。例如，如果想让类构造器在缺少 `new` 的情况下能够工作，你可以使用 `apply` 陷阱函数来创建一个新实例。此处有个例子：

```
class Person {
    constructor(name) {
        this.name = name;
    }
}

let PersonProxy = new Proxy(Person, {
        apply: function(trapTarget, thisArg, argumentList) {
            return new trapTarget(...argumentList);
        }
    });


let me = PersonProxy("Nicholas");
console.log(me.name);                   // "Nicholas"
console.log(me instanceof Person);      // true
console.log(me instanceof PersonProxy); // true
```

`PersonProxy` 对象是 `Person` 类构造器的一个代理。类构造器实际上也是函数，因此在使用代理时它的行为就像函数一样。 `apply` 陷阱函数重写了默认的行为，返回 `trapTarget` （这里等于 `Person` ）的一个实例，此代码使用 `trapTarget` 以保证通用性，避免了手动指定特定的类。此处还使用了扩展运算符，将 `argumentList` 展开并传递给 `trapTarget` 方法。在没有使用 `new` 的情况下调用 `PersonProxy()` ，获得了 `Person` 的一个新实例；而若你试图不使用 `new` 去调用 `Person()` ，构造器仍然会抛出错误。创建一个可被调用的类构造器，只有使用代理才能做到。

可被撤销的代理

在被创建之后，代理通常就不能再从目标对象上被解绑。本章之前的例子都使用了不可被撤销的代理，但有的情况下你可能想撤销一个代理以便让它不能再被使用。当你想通过公共接口向外提供一个安全的对象，并且要求要随时都能切断对某些功能的访问，这种情况下可被撤销的代理就会非常有用。

你可以使用 `Proxy.revocable()` 方法来创建一个可被撤销的代理，该方法接受的参数与 `Proxy` 构造器的相同：一个目标对象、一个代理处理器，而返回值是包含下列属性的一个对象：

1. `proxy` ：可被撤销的代理对象；
2. `revoke` ：用于撤销代理的函数。

当 `revoke()` 函数被调用后，就不能再对该 `proxy` 对象进行更多操作，任何与该代理对象交互的意图都会触发代理的陷阱函数，从而抛出一个错误。例如：

```
let target = {
    name: "target"
};

let { proxy, revoke } = Proxy.revocable(target, {});

console.log(proxy.name);        // "target"

revoke();

// 抛出错误
console.log(proxy.name);
```

这个例子创建了一个可被撤销的代理，它对 `Proxy.revocable()` 方法返回的对象进行了解构赋值，把同名属性的值赋给了 `proxy` 与 `revoke` 变量。此时 `proxy` 对象可以像一个不可被撤销的代理那样被使用，于是 `proxy.name` 属性的值就是 `"target"` ，因为它直接传递了 `target.name` 的值。然而一旦 `revoke()` 函数被调用， `proxy` 就不再是可用的代理对象，之后试图访问 `proxy.name` 会抛出错误，同时其他对于 `proxy` 对象的操作也都会触发陷阱函数。

解决数组的问题

在本章开始时，我解释了为何在 ES6 之前开发者无法准确模拟 JS 数组的行为。而代理与反射接口则允许你创建这样一种对象：在属性被添加或删除时，它的行为与内置数组类型的行为相同。为了唤醒你的记忆，此处有个例子展示了代理所要模拟的行为：

```
let colors = ["red", "green", "blue"];

console.log(colors.length);         // 3

colors[3] = "black";

console.log(colors.length);         // 4
console.log(colors[3]);             // "black"

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
```

这个例子可以体现出两个特别重要的行为特性：

1. 当 `colors[3]` 被赋值时， `length` 属性被自动增加到 4 ；
2. 当 `length` 属性被设置为 2 时，数组的最后两个元素被自动移除了。

当想要重现内置数组的工作方式时，仅需模拟这两个行为即可。接下来的几小节将会介绍如何正确地将一个对象模拟为数组。

检测数组的索引

必须始终牢记：对于数组来说，为整数属性赋值是一种特殊情况，不同于对非整数的键的处理。在如何判断一个属性键是否为数组的索引方面，ES6 规范给出了指南：

> 对于名为 `P` 的一个字符串属性名称来说，当且仅当 `ToString(ToUint32(P))` 等于 `P` 、并且 `ToUint32(P)` 不等于 $2^{32} - 1$ 时，它才能被用作数组的索引。

这个操作可以用下述的 JS 代码来实现：

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}
```

`toUint32()` 函数使用规范中描述的算法，将给定值转换为一个无符号的 32 位整数。`isArrayIndex()` 函数首先将键值转换为一个 uint32 数，并执行了比较操作来判断该键是否能够作为数组的索引。借助这两个工具函数，你就可以开始实现一个对象来模拟内置数组。

在添加新元素时增加长度属性

你可能已经注意到：数组上述两个特殊行为都依赖于对属性的赋值，这就意味着你只需要使用 `set` 代理陷阱来达成这两个行为。首先，下面的例子实现了第一个行为，即：当一个大于 `length - 1` 的数组索引被使用时， `length` 属性需要被增加。

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

function createMyArray(length=0) {
    return new Proxy({ length }, {
        set(trapTarget, key, value) {

            let currentLength = Reflect.get(trapTarget, "length");

            // 特殊情况
            if (isArrayIndex(key)) {
                let numericKey = Number(key);

                if (numericKey >= currentLength) {
                    Reflect.set(trapTarget, "length", numericKey + 1);
                }
            }

            // 无论键的类型是什么，都要执行这行代码
            return Reflect.set(trapTarget, key, value);
        }
    });
}

let colors = createMyArray(3);
console.log(colors.length);         // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";

console.log(colors.length);         // 3

colors[3] = "black";

console.log(colors.length);         // 4
console.log(colors[3]);             // "black"
```

这个例子使用了 `set` 代理陷阱对数组索引的设置操作进行拦截。若该键能够作为数组索引，由于传入的键值始终都是字符串，那么就需要将其转换为一个数值；接下来，如果该数值大于或等于当前的 `length` 属性值，那么要把 `length` 属性值增加到比该数值多 1 （如在索引位置 3 设置一个项，则 `length` 属性必须是 4 ）；最后通过 `Reflect.set()` 来调用属性的默认设置操作，以便让对应属性接收到指定的值。

使用值为 3 的 `length` 参数调用 `createMyArray()` 函数，初始化了一个定制数组，接下来立刻将三个项添加到该数组内。数组的 `length` 属性一直保持为 3 ，直到 `"black"` 值被赋值到索引 3 的位置，此时 `length` 属性就变成了 4 。

这样就成功模拟了第一个行为，该继续处理第二个行为了。

在减少长度属性时移除元素

仅当数组索引值大于或等于 `length` 属性值时，所需模拟的第一个数组行为才会被使用。而相反的，在将 `length` 属性值设置得比之前更小的时候，才需要使用第二个行为并移除数组的元素。此时不仅需要修改 `length` 属性的值，还需要移除所有不应再保留的元素。例如，若数组的 `length` 属性从 4 被设置为 2 ，则位置 2 与位置 3 的项就需要被移除。你可以像处理第一个行为那样，在 `set` 代理陷阱中完成这个操作。下面再次使用了前一段代码，并增加了 `createMyArray` 方法：

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

function createMyArray(length=0) {
    return new Proxy({ length }, {
        set(trapTarget, key, value) {

            let currentLength = Reflect.get(trapTarget, "length");

            // 特殊情况
            if (isArrayIndex(key)) {
                let numericKey = Number(key);

                if (numericKey >= currentLength) {
                    Reflect.set(trapTarget, "length", numericKey + 1);
                }
            } else if (key === "length") {

                if (value < currentLength) {
                    for (let index = currentLength - 1; index >= value; index--) {
                        Reflect.deleteProperty(trapTarget, index);
                    }
                }

            }

            // 无论键的类型是什么，都要执行这行代码
            return Reflect.set(trapTarget, key, value);
```

```
        }
    });
}

let colors = createMyArray(3);
console.log(colors.length);         // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
colors[3] = "black";

console.log(colors.length);         // 4

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
console.log(colors[0]);             // "red"
```

此代码中的 `set` 陷阱函数会检查键的值是否为 `"length"` ，以便正确地调整对象的剩余项。如果是，则使用 `Reflect.get()` 方法来获取当前的长度值，并与新值作比较。如果新值小于当前值，将会使用一个 `for` 循环来删除对象上所有不应再被保留的属性，该循环从当前数组长度（ `currentLength` ）的位置向前删除每个属性，直到触及新的数组长度（ `value` ）为止。

该例子先向 `colors` 对象中添加了四个颜色，再将其 `length` 属性设置为 2 ，结果移除了位置 2 与位置 3 的项，这样在试图访问这两个项的时候就会得到 `undefined` 。而位置 0 与位置 1 的项仍然可被访问。

两个行为都实现之后，你就可以轻易创建一个对象来模拟内置数组的行为。然而使用函数来做这些事并不可取，最好将其封装为一个类，因此下一步就是使用类来实现这些功能。

实现 MyArray 类

创建一个使用代理的类的最简单方式，就是照常定义一个类但从构造器中返回一个代理。这种方式下，该类被实例化时返回的对象就是代理，而不是该类的实例（实例即构造器内部的 `this` 值）。代理会像实例一样被返回，而实例此时就变成了该代理的目标对象。实例将会是完全私有的，无法被直接访问，不过你可以使用代理去间接访问它。

这里有一个从类构造器返回代理的简单范例：

```
class Thing {
    constructor() {
        return new Proxy(this, {});
    }
}

let myThing = new Thing();
console.log(myThing instanceof Thing);      // true
```

在这个例子中， `Thing` 类从它的构造器中返回了一个代理，该代理的目标对象是构造器被调用时其内部的 `this` 。这意味着虽然 `myThing` 对象是调用 `Thing` 构造器创建的，但它实际上是一个代理对象。由于此代理将行为直接传递给它的目标对象， 因而 `myThing` 仍然可以被认定为 `Thing` 类的一个实例，并且让代理在使用 `Thing` 类时完全透明。

知道了这些，使用代理来创建一个定制的数组类就相当简单了。它的实现代码与“在减少长度属性时移除元素”那个小节的代码非常接近，使用了相同的代理代码，但这次是在类的构造器中使用它。此处有个完整的范例：

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

class MyArray {
    constructor(length=0) {
        this.length = length;

        return new Proxy(this, {
            set(trapTarget, key, value) {

                let currentLength = Reflect.get(trapTarget, "length");

                // 特殊情况
                if (isArrayIndex(key)) {
                    let numericKey = Number(key);

                    if (numericKey >= currentLength) {
                        Reflect.set(trapTarget, "length", numericKey + 1);
                    }
                } else if (key === "length") {

                    if (value < currentLength) {
                        for (let index = currentLength - 1; index >= value; index--) {
                            Reflect.deleteProperty(trapTarget, index);
                        }
```

```
                    }

                }

                // 无论键的类型是什么，都要执行这行代码
                return Reflect.set(trapTarget, key, value);
            }
        });

    }
}


let colors = new MyArray(3);
console.log(colors instanceof MyArray);     // true

console.log(colors.length);         // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
colors[3] = "black";

console.log(colors.length);         // 4

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
console.log(colors[0]);             // "red"
```

这段代码创建了一个 `MyArray` 类，并从构造器中返回了一个代理。在构造器中，添加了 `length` 属性（使用传入的值进行初始化，或者在值未提供的情况下使用默认的 0 ），然后创建并返回了一个代理。这让 `colors` 变量看起来就像是 `MyArray` 类的一个实例，并且实现了数组的两个关键行为。

虽然从类构造器中返回一个代理是很容易的，但这意味着每个实例都会创建一个新的代理。不过你可以将代理对象作为原型使用，这样就可以在所有实例上共享一个代理。

将代理对象作为原型使用

代理对象可以被作为原型使用，但这么做会比本章前面的例子更复杂一些。在把代理对象作为原型时，仅当操作的默认行为会按惯例追踪原型时，代理陷阱才会被调用，这就限制了代理对象作为原型时的能力。考虑这个例子：

```
let target = {};
let newTarget = Object.create(new Proxy(target, {

    // 永远不会被调用
    defineProperty(trapTarget, name, descriptor) {

        // 如果被调用就会引发错误
        return false;
    }
}));

Object.defineProperty(newTarget, "name", {
    value: "newTarget"
});

console.log(newTarget.name);                    // "newTarget"
console.log(newTarget.hasOwnProperty("name"));  // true
```

一个代理被作为原型创建了 `newTarget` 对象。将 `target` 作为代理的目标对象，由于该代理是透明的， `target` 就有效地成为了 `newTarget` 的原型。此时，只有当 `newTarget` 将操作传递给 `target` 的时候，代理陷阱才会被调用。

`Object.defineProperty()` 方法在 `newTarget` 上被调用，创建了一个自有属性 `name` 。定义对象属性的操作并不会按惯例追踪对象原型，因此代理上的 `defineProperty` 陷阱函数永远不会被调用，于是 `name` 属性就被添加到了 `newTarget` 对象上，成为它的一个自有属性。

尽管在把代理对象作为原型时会受到严重限制，但仍然存在几个很有用的陷阱函数。

在原型上使用 get 陷阱函数

当内部方法 `[[Get]]` 被调用以读取属性时，该操作首先会查找对象的自有属性；如果指定名称的属性没有找到，则会继续在对象的原型上进行属性查找；这个流程会一直持续到没有原型可供查找为止。

得益于这个流程，若你设置了一个 `get` 代理陷阱，则只有在对象不存在指定名称的自有属性时，该陷阱函数才会在对象的原型上被调用。当所访问的属性无法保证存在时，你可以使用 `get` 陷阱函数来阻止预期外的行为。下例创建了一个对象，当你尝试去访问一个不存在的属性时，它会抛出错误：

```
let target = {};
let thing = Object.create(new Proxy(target, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
}));

thing.name = "thing";

console.log(thing.name);        // "thing"

// 抛出错误
let unknown = thing.unknown;
```

这段代码创建了一个将代理作为原型的 `thing` 对象。当 `thing` 对象中不存在指定键的时候， `get` 陷阱函数就会抛出错误。在读取 `thing.name` 时，因为该属性存在于 `thing` 对象中， `get` 陷阱函数没有被调用；而当读取不存在的 `thing.unknown` 属性时，才调用了 `get` 陷阱函数。

当最后一行代码执行时， `unknown` 并不是 `thing` 的自有属性，因此查找操作延续到了它的原型上，于是 `get` 陷阱函数抛出了一个错误。这种自定义行为对 JS 来说是非常有用的，因为它能够让 JS 像其他语言那样、在访问不存在的属性时抛出错误，而不是静默地返回 `undefined` 。

`trapTarget` 与 `receiver` 是不同的对象，这对理解本例是非常重要的。当代理被用作原型时， `trapTarget` 是原型对象自身，而 `receiver` 则是实例对象。这意味着在本例中， `trapTarget` 等于 `target` ，而 `receiver` 则等于 `thing` 。这就使得你既能访问代理的原始目标对象，也能访问操作将要涉及的对象。

在原型上使用 set 陷阱函数

内部方法 `[[Set]]` 同样会查找对象的自有属性，并在必要时继续对该对象的原型进行查找。当你对一个对象属性进行赋值时，如果指定名称的自有属性存在，值就会被赋在该属性上；而若该自有属性不存在，则会继续检查对象的原型。微妙之处在于：尽管赋值操作在原型上继续进行，但默认情况下它会在对象实例（而非原型）上创建一个新的属性用于赋值，无论同名属性是否存在于原型上。

为了更好地了解 `set` 陷阱函数何时会在原型上被调用、而何时不会，可研究下面这个展示了默认行为的示例：

```
let target = {};
let thing = Object.create(new Proxy(target, {
    set(trapTarget, key, value, receiver) {
        return Reflect.set(trapTarget, key, value, receiver);
    }
}));

console.log(thing.hasOwnProperty("name"));      // false

// 触发了 `set` 代理陷阱
thing.name = "thing";

console.log(thing.name);                        // "thing"
console.log(thing.hasOwnProperty("name"));      // true

// 没有触发 `set` 代理陷阱
thing.name = "boo";

console.log(thing.name);                        // "boo"
```

在本例中， `target` 对象起初未拥有任何自有属性。 `thing` 对象把一个代理作为自身的原型，并定义了一个 `set` 陷阱函数来捕获任意创建新属性的操作。当 `thing.name` 被赋值为 `"thing"` 时，因为 `thing` 对象并不存在一个名为 `name` 的自有属性， `set` 代理陷阱就被调用。在 `set` 陷阱函数中， `trapTarget` 参数等于 `target` ，而 `receiver` 参数则等于 `thing` 。你可以将 `receiver` 作为第四个参数传递给 `Reflect.set()` 方法来实现默认的行为，最终一个新的属性就在 `thing` 对象上被创建了。

一旦 `thing` 对象的 `name` 属性被创建完毕，将 `thing.name` 另设为其他值就不会再触发原型上 `set` 代理陷阱，因为此时 `name` 变成了自有属性， `[[Set]]` 操作便不会再继续查找原型了。

在原型上使用 has 陷阱函数

可以回忆一下， `has` 陷阱函数会拦截对象上 `in` 运算符的使用。 `in` 运算符首先查找对象上指定名称的自有属性；如果不存在同名自有属性，则会继续查找对象的原型；如果原型上也不存在同名自有属性，那么就会沿着原型链一直查找下去，直到找到该属性、或者没有更多原型可供查找时为止。

`has` 陷阱函数只在原型链查找触及原型对象的时候才会被调用。当使用代理作为原型时，这只会在指定名称的自有属性不存在时发生。例如：

```
let target = {};
let thing = Object.create(new Proxy(target, {
    has(trapTarget, key) {
        return Reflect.has(trapTarget, key);
    }
}));

// 触发了 `has` 代理陷阱
console.log("name" in thing);                   // false

thing.name = "thing";

// 没有触发 `has` 代理陷阱
console.log("name" in thing);                   // true
```

此代码在 `thing` 的原型上创建了一个 `has` 代理陷阱。 `has` 陷阱函数并没有像 `get` 或 `set` 陷阱函数那样传递一个 `receiver` 参数，因为当 `in` 运算符被使用时，对原型的查找是自动的。相反的， `has` 陷阱函数只能对 `trapTarget` 参数进行操作，该参数等于 `target` 。本例中第一次使用 `in` 运算符的时候，由于 `thing` 并不存在自有属性 `name` ，于是 `has` 陷阱函数就被调用了。而当 `thing.name` 被赋值之后，再次使用 `in` 运算符， `has` 陷阱函数则不会被调用，因为操作在找到 `thing` 的自有属性 `name` 后便已停止。

这里的原型范例都围绕着使用 `Object.create()` 方法创建的对象。然而若你想创建一个以代理为原型的对象，流程会有些不同。

将代理作为类的原型

由于类的 `prototype` 属性是不可写入的，因此不能直接修改类，将代理用作它的原型。然而你可以使用一点变通手段，利用继承来创建一个把代理作为自身原型的类。首先你需要使用构造器函数创建一个 ES5 风格的类定义。你可以将原型改写为一个代理，此处有个例子：

```
function NoSuchProperty() {
    // empty
}

NoSuchProperty.prototype = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

let thing = new NoSuchProperty();

// 由于 `get` 代理陷阱而抛出了错误
let result = thing.name;
```

`NoSuchProperty` 函数代表了将会被用于继承的基础类。此函数的 `prototype` 属性不存在任何限制，因此你可以将其改写为一个代理，其中 `get` 陷阱函数被用于在属性缺失时抛出错误。 `thing` 对象被创建为 `NoSuchProperty` 类的一个实例，当访问不存在的 `name` 属性时，错误就被抛出。

下一步是创建一个继承 `NoSuchProperty` 的类。你可以简单使用第九章介绍过的 `extends` 语法，来将代理引入该类的原型链，就像这样：

```
function NoSuchProperty() {
    // empty
}

NoSuchProperty.prototype = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

class Square extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

let shape = new Square(2, 6);

let area1 = shape.length * shape.width;
console.log(area1);                         // 12

// 由于 "wdth" 不存在而抛出了错误
let area2 = shape.length * shape.wdth;
```

`Square` 类继承了 `NoSuchProperty` 类，因此该代理就被加入了 `Square` 类的原型链。随后 `shape` 对象被创建为 `Square` 类的一个实例，让它拥有两个属性： `length` 与 `width` 。由于这两个属性存在， `get` 陷阱函数就永远不会被调用，因此能够成功读取它们的值。只有访问 `shape` 上不存在的属性时（例如此处的 `shape.wdth` 拼写错误），才触发了 `get` 陷阱函数并导致错误被抛出。

这证明了该代理存在于 `shape` 的原型链中，但这可能并不明显，因为该代理不是 `shape` 的直接原型。事实上，该代理需要用两步才能从 `shape` 的原型链上被找到。你可以修改前面的例子来更清晰地领会这一点：

```
function NoSuchProperty() {
    // 空
}

// 对于将要用作原型的代理，存储对其的一个引用
let proxy = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

NoSuchProperty.prototype = proxy;

class Shape extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

let shape = new Square(2, 6);

let shapeProto = Object.getPrototypeOf(shape);

console.log(shapeProto === proxy);                 // false

let secondLevelProto = Object.getPrototypeOf(shapeProto);

console.log(secondLevelProto === proxy);           // true
```

这个版本的代码将代理存储在一个名为 `proxy` 的变量中，以便之后可以简单识别。 `shape` 的原型是 `Square.prototype` ，它并不是一个代理。然而 `Shape.prototype` 的原型却是一个从 `NoSuchProperty` 继承下来的代理。

继承行为在原型链上增加了一步，明白这一点很重要，因为在 `proxy` 变量上调用 `get` 陷阱函数的操作也需要多进行一步。如果欲使用的属性存在于 `Shape.prototype` 上，那么这就会防止 `get` 代理陷阱被调用，正如此例：

```
function NoSuchProperty() {
    // empty
}

NoSuchProperty.prototype = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

class Square extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

let shape = new Square(2, 6);

let area1 = shape.length * shape.width;
console.log(area1);                         // 12

let area2 = shape.getArea();
console.log(area2);                         // 12

// 由于 "wdth" 不存在而抛出了错误
let area3 = shape.length * shape.wdth;
```

此处的 `Square` 类拥有一个 `getArea()` 方法，该方法被自动添加到 `Square.prototype` 上，因此当 `shape.getArea()` 被调用时，对于 `getArea()` 方法的查找从 `shape` 实例上开始，并延续到它的原型上。由于在原型上找到了 `getArea()` 方法，查找就停止了，代理也没有被调用。在本例的条件下，这正是你想要的行为，而不应当在 `getArea()` 被调用时抛出错误。

尽管使用了一点额外的代码来创建一个类，才让代理存在于该类的原型链上，但当你确实需要这样的功能时，这种付出仍然是值得的。

总结

在 ES6 之前，数组之类的特定对象会显示出一些非常规的、无法被开发者复制的行为，而代理的出现改变了这种情况。代理允许你为一些 JS 底层操作自行定义非常规行为，因此你就可以通过代理陷阱来复制 JS 内置对象的所有行为。在各种不同操作发生时（例如对于 `in` 运算符的使用），这些代理陷阱会在后台被调用。

反射接口也是在 ES6 中引入的，允许开发者为每个代理陷阱实现默认的行为。每个代理陷阱在 ES6 的另一个新特性 `Reflect` 对象上都有一个同名的对应方法。将代理陷阱与反射接口方法结合使用，就可以在特定条件下让一些操作有不同的表现，有别于默认的内置行为。

可被撤销的代理是一种特殊的代理，可以使用 `revoke()` 函数去有效禁用。 `revoke()` 函数终结了代理的所有功能，因此在它被调用之后，所有与代理属性交互的意图都会导致抛出错误。第三方开发者可能需要在一定时间内获取特定对象的使用权，在这种场合，可被撤销的代理对应用的安全性来说就非常重要。

尽管直接使用代理是最有力的使用方式，但你也可以把代理用作其他对象的原型。但只有很少的代理陷阱能在作为原型的代理上被有效使用，包括 `get` 、 `set` 与 `has` 这几个，这方面的用例也因此变得十分有限。

第十三章 用模块封装代码

JS "共享一切"的代码加载方式是该语言混乱且最易出错的方面之一。其他语言使用包（ package ）之类的概念来定义代码的作用域，然而在 ES6 之前，一个应用的每个 JS 文件所定义的所有内容都由全局作用域共享。当 web 应用变得更加复杂、需要使用越来越多的 JS 代码时，这种方式导致了诸多问题，例如命名冲突、安全问题等。 ES6 的设计目标之一就是要解决作用域问题，并让 JS 应用变得更有条理。这便是模块的切入点。

何为模块？

模块（ **Modules** ）是使用不同方式加载的 JS 文件（与 JS 原先的脚本加载方式相对）。这种不同模式很有必要，因为它与脚本（ **script** ）有大大不同的语义：

1. 模块代码自动运行在严格模式下，并且没有任何办法跳出严格模式；
2. 在模块的顶级作用域创建的变量，不会被自动添加到共享的全局作用域，它们只会在模块顶级作用域的内部存在；
3. 模块顶级作用域的 `this` 值为 `undefined` ；

4. 模块不允许在代码中使用 HTML 风格的注释（这是 JS 来自于早期浏览器的历史遗留特性）；
5. 对于需要让模块外部代码访问的内容，模块必须导出它们；
6. 允许模块从其他模块导入绑定。

这些差异乍一看似乎很小，但它们代表了 JS 代码加载与执行方面的显著改变，我将在整章中对其进行论述。模块的真实力量是按需导出与导入代码的能力，而不用将所有内容放在同一个文件内。对于导出与导入的清楚理解，是辨别模块与脚本差异的基础。

基本的导出

你可以使用 `export` 关键字将已发布代码部分公开给其他模块。最简单方法就是将 `export` 放置在任意变量、函数或类声明之前，从模块中将它们公开出去，就像这样：

```
// 导出数据
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

// 导出函数
export function sum(num1, num2) {
    return num1 + num1;
}

// 导出类
export class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
}

// 此函数为模块私有
function subtract(num1, num2) {
    return num1 - num2;
}

// 定义一个函数……
function multiply(num1, num2) {
    return num1 * num2;
}

// ……稍后将其导出
export { multiply };
```

此例中有几点需要注意。首先，除了 `export` 关键字之外，每个声明都与正常形式完全一样。每个被导出的函数或类都有名称，这是因为导出的函数声明与类声明必须要有名称。你不能使用这种语法来导出匿名函数或匿名类，除非使用了 `default` 关键字（在“模块的默认

值”一节会论述）。

其次，细看一下 `multiply()` 函数，它并没有在定义时被导出。这是因为你不仅能导出声明，还可以导出引用（即代码最后一行）。

最后请注意，此例并未导出 `subtract()` 函数。此函数在模块外部不可访问，因为任意没有被显式导出的变量、函数或类都会在模块内保持私有。

基本的导入

一旦你有了包含导出的模块，就能在其他模块内使用 `import` 关键字来访问已被导出的功能。 `import` 语句有两个部分，一是需要导入的标识符，二是需导入的标识符的来源模块。此处是导入语句的基本形式：

```
import { identifier1, identifier2 } from "./example.js";
```

在 `import` 之后的花括号指明了从给定模块导入对应的绑定， `from` 关键字则指明了需要导入的模块。模块由一个表示模块路径的字符串（被称为模块说明符， **module specifier** ）来指定。在浏览器环境中导入模块，使用与 `<script>` 元素相同的路径格式，这表示你必须在其中包含文件扩展名。而另一方面 Node.js 则遵循了它的传统惯例，基于文件系统前缀来分辨本地文件与包（ package ），例如， `example` 代表一个包，而 `./example.js` 则代表一个本地文件。

> 导入绑定的列表看起来与对象解构相似，但实则并无关联。

当从模块导入了一个绑定时，该绑定表现得就像使用了 `const` 的定义。这意味着你不能再定义另一个同名变量（包括导入另一个同名绑定），也不能在对应的 `import` 语句之前使用此标识符（也就是要受暂时性死区限制），更不能修改它的值。

导入单个绑定

对于“基本的导入”小节的第一个例子，先假设它位于一个文件名为 `example.js` 的模块内。你能用多种方式来导入并使用来自该模块的绑定。例如，你可以仅导入一个标识符：

```
// 单个导入
import { sum } from "./example.js";

console.log(sum(1, 2));     // 3

sum = 1;        // 出错
```

尽管 `example.js` 并非只导出了一个 `sum()` 函数，但本例仅仅导入了此函数。假若你尝试给 `sum` 赋一个新值，由于不允许对已导入的绑定重新赋值，于是就会导致错误。

> 要确保在导入的文件名前面使用 `/` 、 `./` 或 `../` ，以便在浏览器与 Node.js 之间保持良好兼容性。

导入多个绑定

如果你想从 example 模块导入多个绑定，你可以像下面这样显式的列出它们：

```
// 多个导入
import { sum, multiply, magicNumber } from "./example.js";
console.log(sum(1, magicNumber));    // 8
console.log(multiply(1, 2));         // 2
```

此处从 example 模块导入了三个绑定： `sum` 、 `multiply` 与 `magicNumber` ，之后便可以使用它们，仿佛它们是在当前模块中被定义的。

完全导入一个模块

还有一种特殊情况，即允许你将整个模块当作单一对象进行导入，该模块的所有导出都会作为对象的属性存在。例如：

```
// 完全导入
import * as example from "./example.js";
console.log(example.sum(1,
        example.magicNumber));          // 8
console.log(example.multiply(1, 2));    // 2
```

在此代码中， `example.js` 中所有导出的绑定都被加载到一个名为 `example` 的对象中，具名导出（ `sum()` 函数、 `multiple()` 函数与 `magicNumber` ）都成为 `example` 的可用属性。这种导入格式被称为命名空间导入（ **namespace import** ），这是因为该 `example` 对象并不存在于 `example.js` 文件中，而是作为一个命名空间对象被创建使用，其中包含了 `example.js` 的所有导出成员。

然而要记住，无论你对同一个模块使用了多少次 `import` 语句，该模块都只会被执行一次。在导出模块的代码执行之后，已被实例化的模块就被保留在内存中，并随时都能被其他 `import` 所引用。研究以下例子：

```
import { sum } from "./example.js";
import { multiply } from "./example.js";
import { magicNumber } from "./example.js";
```

尽管此处的模块使用了三个 `import` 语句，但 `example.js` 只会被执行一次。若同一个应用中的其他模块打算从 `example.js` 导入绑定，则那些模块都会使用这段代码中所用的同一个模块实例。

> 模块语法的限制（Module Syntax Limitations）
>
> `export` 与 `import` 都有一个重要的限制，那就是它们必须被用在其他语句或表达式的外部。例如，以下代码有语法错误：
>
> ```
> if (flag) {
>   export flag;    // 语法错误
> }
> ```
>
> 此处的 `export` 语句位于一个 `if` 语句内部，这是不被许可的。导出语句不能是有条件的，也不能以任何方式动态使用。原因之一是模块语法需要让 JS 能静态判断需要导出什么，正因为此，你只能在模块的顶级作用域使用 `export` 。
>
> 类似的，你不能在一个语句内部使用 `import` ，也只能将其用在顶级作用域。这意味着以下代码也有语法错误：
>
> ```
> function tryImport() {
>   import flag from "./example.js";    // 语法错误
> }
> ```
>
> 出于与不能动态导出绑定相同的原因，你也不能动态导入绑定。 `export` 与 `import` 关键字被设计为静态的，以便让诸如文本编辑器之类的工具能轻易判断模块有哪些信息可用。

导入绑定的一个微妙怪异点

ES6 的 `import` 语句为变量、函数与类创建了只读绑定，而不像普通变量那样简单引用了原始绑定。尽管导入绑定的模块无法修改绑定的值，但负责导出的模块却能做到这一点。例如，假设你想要使用以下模块：

```
export var name = "Nicholas";
export function setName(newName) {
    name = newName;
}
```

当你导入了这两个绑定后， `setName()` 函数还可以改变 `name` 的值：

```
import { name, setName } from "./example.js";

console.log(name);       // "Nicholas"
setName("Greg");
console.log(name);       // "Greg"

name = "Nicholas";       // error
```

调用 `setName("Greg")` 会回到导出 `setName()` 的模块内部，并在那里执行，从而将 `name` 设置为 `"Greg"` 。注意这个变化会自动反映到所导入的 `name` 绑定上，这是因为绑定的 `name` 是导出的 `name` 标识符的本地名称，二者并非同一个事物。

> 译注：对本小节内容进行补充说明
>
> ```
> let a = 1;
> let b = a;
> console.log(a);        // 1
> console.log(b);        // 1
> a = 2;
> console.log(a);        // 2
> console.log(b);        // 1
> ```
>
> 在此代码中，变量 `b` 开始时貌似对变量 `a` 进行了一个“引用”，但只是将 `a` 的值拷贝了一份。如果对变量 `a` 的值进行修改，变量 `b` 的值是不会随着变化的。
>
> 而在范例中的模块导入与导出，外部模块导入的 `name` 变量与在 `example.js` 模块内部的 `name` 变量对比，前者是对于后者的只读引用，会始终反映出后者的变化。就算后者的值在负责导出的模块中发生了变化，这种绑定关系也不会被破坏。

重命名导出与导入

有时，你可能并不想使用从模块中导出的变量、函数或类的原始名称。幸好，你可以更改导出的名称，无论是在导出过程中，还是在导入过程中。

前一种情况下，假设你想用不同的名称来导出一个函数，你可以使用 `as` 关键字来指定新的名称，以便在模块外部用此名称指代目标函数：

```
function sum(num1, num2) {
    return num1 + num2;
}

export { sum as add };
```

此处的 `sum()` 函数被作为 `add()` 导出，前者是本地名称（ **local name** ），后者则是导出名称（ **exported name** ）。这意味着当另一个模块要导入此函数时，它必须改用 `add` 这个名称：

```
import { add } from "./example.js";
```

假若模块导入函数时想使用另一个名称，同样也可以用 `as` 关键字：

```
import { add as sum } from "./example.js";
console.log(typeof add);            // "undefined"
console.log(sum(1, 2));             // 3
```

此代码导入了 `add()` 函数，并使用了导入名称（ **import name** ）将其重命名为 `sum()` （本地名称）。这意味着在此模块中并不存在名为 `add` 的标识符。

模块的默认值

模块语法确实为从模块中导出或导入默认值进行了优化，而这一模式在其他模块系统中非常普遍，例如在 CommonJS （在浏览器之外运行 JS 的另一种模块规范）中。模块的默认值（ **default value** ）是使用 `default` 关键字所指定的单个变量、函数或类，而你在每个模块中只能设置一个默认导出，将 `default` 关键字用于多个导出会是语法错误。

导出默认值

以下是使用 `default` 关键字的一个简单例子：

```
export default function(num1, num2) {
    return num1 + num2;
}
```

此模块将一个函数作为默认值进行了导出， `default` 关键字标明了这是一个默认导出。此函数并不需要有名称，因为它就代表这个模块自身。

你也能在 `export default` 后面放置一个标识符，以指定默认的导出，正如：

```
function sum(num1, num2) {
    return num1 + num2;
}

export default sum;
```

此处 `sum()` 函数先被定义了，随后它作为模块的默认值被导出。若默认值需要计算才能得出，你或许会选择这种方式。

将标识符作为默认导出来指定的第三种方式，是使用重命名语法，如下：

```
function sum(num1, num2) {
    return num1 + num2;
}

export { sum as default };
```

`default` 标识符有特别含义，既作为重命名导出，又标明了模块需要使用的默认值。由于 `default` 在 JS 中是一个关键字，它就不能被用作变量、函数或类的名称（但它可以被用作属性名称）。因此使用 `default` 来重命名一个导出是个特例，与非默认导出的语法保持了一致性。若你想用单个语句一次性进行多个导出，并要求包含默认导出，这种语法就非常有用。

导入默认值

你可以使用如下语法来从一个模块中导入默认值：

```
// 导入默认值
import sum from "./example.js";

console.log(sum(1, 2));     // 3
```

这个导入语句从 `example.js` 模块导入了其默认值。注意此处并未使用花括号，与之前在非默认的导入中看到的不同。本地名称 `sum` 被用于代表目标模块所默认导出的函数。这种语法是最简洁的，而 ES6 的标准制定者也期待它成为在网络上进行导入的主要形式，这样你就能导入已存在的对象。

对于既导出了默认值、又导出了一个或更多非默认的绑定的模块，你可以使用单个语句来导入它的所有导出绑定。例如，假设你有这么一个模块：

```
export let color = "red";

export default function(num1, num2) {
    return num1 + num2;
}
```

你可以像下面这样使用 `import` 语句，来同时导入 `color` 以及作为默认值的函数：

```
import sum, { color } from "./example.js";

console.log(sum(1, 2));     // 3
console.log(color);         // "red"
```

逗号将默认的本地名称与非默认的名称分隔开，后者仍旧被花括号所包裹。要记住在 `import` 语句中默认名称必须位于非默认名称之前。

如同导出默认值，你也能使用重命名语法进行默认值的导入：

```
// 等价于上个例子
import { default as sum, color } from "example";

console.log(sum(1, 2));      // 3
console.log(color);          // "red"
```

在此代码中，默认的导出（ `default` ）被重命名为 `sum` ，并且附加的 `color` 导出也被一并导入了。此例与前面的例子是等效的。

绑定的再导出

也许有时你会想将当前模块已导入的内容重新再导出（例如，假设要用几个小模块来创建一个库）。你能使用本章已描述过的模式来将已导入的值再导出，就像这样：

```
import { sum } from "./example.js";
export { sum }
```

此方法能奏效，但还可以使用单个语句来完成相同任务：

```
export { sum } from "./example.js";
```

这种形式的 `export` 会进入指定模块查看 `sum` 的定义，随后将其导出。当然，你也可以选择将一个值用不同名称导出：

```
export { sum as add } from "./example.js";
```

此处，从 `"./example.js"` 导入的 `sum` 随后以 `add` 的名称被导出了。

若你想将来自另一个模块的所有值完全导出，可以使用星号（ `*` ）模式：

```
export * from "./example.js";
```

使用完全导出，就可以导出目标模块的默认值及其所有具名导出，但这可能影响你从当前模块所能导出的值。例如，假设 `example.js` 具有一个默认导出，当你使用这种语法时，你就无法为当前模块另外再定义一个默认导出。

无绑定的导入

有些模块也许没有进行任何导出，相反只是修改全局作用域的对象。尽管这种模块的顶级变量、函数或类最终并不会自动被加入全局作用域，但这并不意味着该模块无法访问全局作用域。诸如 `Array` 与 `Object` 之类的内置对象的共享定义在模块内部是可访问的，并且对于这

些对象的修改会反映到其他模块中。

例如，若你想为所有数组添加一个 `pushAll()` 方法，你可以像下面这样定义一个模块：

```
// 没有导出与导入的模块
Array.prototype.pushAll = function(items) {

    // items 必须是一个数组
    if (!Array.isArray(items)) {
        throw new TypeError("Argument must be an array.");
    }

    // 使用内置的 push() 与扩展运算符
    return this.push(...items);
};
```

这是一个有效的模块，尽管此处没有任何导出与导入。此代码可以作为模块或脚本来使用。由于它没有导出任何东西，你可以使用简化的导入语法来执行此模块的代码，而无须导入任何绑定：

```
import "./example.js";

let colors = ["red", "green", "blue"];
let items = [];

items.pushAll(colors);
```

此代码导入并执行了包含 `pushAll()` 的模块，于是 `pushAll()` 就被添加到数组的原型上。这意味着现在 `pushAll()` 在当前模块内的所有数组上都可用。

> 无绑定的导入最有可能被用于创建 polyfill 与 shim （为新语法在旧环境中运行提供向下兼容的方式）。

加载模块

尽管 ES6 定义了模块的语法，但并未定义如何加载它们。这是规范复杂性的一部分，这种复杂性对于实现环境来说是无法预知的。 ES6 未选择给所有 JS 环境努力创建一个有效的单一规范，而只规定了语法，并指定了一个未定义的内部操作 `HostResolveImportedModule` 的抽象加载机制。 web 浏览器与 Node.js 可以自行决定用什么方式实现 `HostResolveImportedModule` ，以便更好契合各自的环境。

在 Web 浏览器中使用模块

即使在 ES6 之前， web 浏览器都有多种方式在 web 应用中加载 JS 。这些可能的脚本加载选择是：

1. 使用 `<script>` 元素以及 `src` 属性来指定代码加载的位置，以便加载 JS 代码文件；
2. 使用 `<script>` 元素但不使用 `src` 属性，来嵌入内联的 JS 代码；
3. 加载 JS 代码文件并作为 Worker （例如 Web Worker 或 Service Worker ）来执行。

为了完全支持模块， web 浏览器必须更新这些机制。相关细节被定义在 HTML 规范中，我将会在本节对其进行概述。

在 script 标签中使用模块

`<script>` 元素默认以脚本而非模块方式来加载 JS 文件，只要 `type` 属性缺失，或者 `type` 属性含有与 JS 对应的内容类型（例如 `"text/javascript"` ）。 `<script>` 元素能够执行内联脚本，也能加载在 `src` 中指定的文件。为了支持模块，添加了 `"module"` 值作为 `type` 的选项。将 `type` 设置为 `"module"` ，就告诉浏览器要将内联代码或是指定文件中的代码当作模块，而不是当作脚本。此处有个简单范例：

```
<!-- load a module JavaScript file -->
<script type="module" src="module.js"></script>

<!-- include a module inline -->
<script type="module">

import { sum } from "./example.js";

let result = sum(1, 2);

</script>
```

此例中第一个 `<script>` 元素使用 `src` 加载了外部模块文件，与加载脚本唯一的区别是将 `type` 指定为 `"module"` 。第二个 `<script>` 元素则包含了一个直接嵌入到网页内的模块， `result` 变量并未被暴露到全局，因为它只在使用 `<script>` 元素定义的这个模块内部存在，因此也没有被添加为 `window` 对象的属性。

正如你所见，在网页中包含模块十分简单，并且类似于包含脚本。然而，在如何加载模块方面有一些区别。

> 你可能已经注意到 `"module"` 并不是与 `"text/javascript"` 相似的内容类型。模块 JS 文件的内容类型与脚本 JS 文件相同，因此不可能依据文件的内容类型将它们完全区别开来。此外，当 `type` 属性无法辨认时，浏览器就会忽略 `<script>` 元素，因此不支持模块的浏览器也就会自动忽略 `<script type="module">` 声明，从而提供良好的向下兼容性。

Web 浏览器中的模块加载次序

模块相对脚本的独特之处在于：它们能使用 `import` 来指定必须要加载的其他文件，以保证正确执行。为了支持此功能， `<script type="module">` 总是自动应用 `defer` 属性。

`defer` 属性是加载脚本文件时的可选项，但在加载模块文件时总是自动应用的。当 HTML 解析到拥有 `src` 属性的 `<script type="module">` 标签时，就会立即开始下载模块文件，但并不会执行它，直到整个网页文档全部解析完为止。模块也会按照它们在 HTML 文件中出现的顺序依次执行，这意味着第一个 `<script type="module">` 总是保证在第二个之前执行，即使其中有些模块不是用 `src` 指定而是包含了内联脚本。例如：

```
<!-- this will execute first -->
<script type="module" src="module1.js"></script>

<!-- this will execute second -->
<script type="module">
import { sum } from "./example.js";

let result = sum(1, 2);
</script>

<!-- this will execute third -->
<script type="module" src="module2.js"></script>
```

这三个 `<script>` 元素依照它们被指定的顺序执行，因此 `module1.js` 保证在内联模块之前执行，而内联模块又保证在 `module2.js` 之前执行。

每个模块可能都用 `import` 导入了一个或多个其他模块，这就让事情变复杂了。这也就是模块为何首先需要被解析，因为这样才能识别所有的 `import` 语句。每个 `import` 语句又会触发一次 fetch （无论是从网络还是从缓存中获取），并且在所有用 `import` 导入的资源被加载与执行完毕之前，没有任何模块会被执行。

所有模块，无论是用 `<script type="module">` 显式包含的，还是用 `import` 隐式包含的，都会依照次序加载与执行。在前面的范例中，完整的加载次序是：

1. 下载并解析 `module1.js` ；
2. 递归下载并解析在 `module1.js` 中使用 `import` 导入的资源；
3. 解析内联模块；
4. 递归下载并解析在内联模块中使用 `import` 导入的资源；
5. 下载并解析 `module2.js` ；
6. 递归下载并解析在 `module2.js` 中使用 `import` 导入的资源。

一旦加载完毕，直到页面文档被完整解析之前，都不会有任何代码被执行。在文档解析完毕后，会发生下列行为：

1. 递归执行 `module1.js` 导入的资源；
2. 执行 `module1.js` ；
3. 递归执行内联模块导入的资源；
4. 执行内联模块；
5. 递归执行 `module2.js` 导入的资源；

6. 执行 `module2.js` 。

注意内联模块除了不必先下载代码之外，与其他两个模块的行为一致，加载 `import` 的资源与执行模块的次序都是完全一样的。

> `<script type="module">` 上的 `defer` 属性总是会被忽略，因为它已经应用了该属性。

Web 浏览器中的异步模块加载

你或许已熟悉了 `<script>` 元素上的 `async` 属性。当配合脚本使用时， `async` 会导致脚本文件在下载并解析完毕后就立即执行。但带有 `async` 的脚本在文档中的顺序却并不会影响脚本执行的次序，脚本总是会在下载完成后就立即执行，而无须等待包含它的文档解析完毕。

`async` 属性也同样能被应用到模块上。在 `<script type="module">` 上使用 `async` 会导致模块的执行行为与脚本相似。唯一区别是模块中所有 `import` 导入的资源会在模块自身被执行前先下载。这保证了模块中所有需要的资源会在模块执行前被下载，你只是不能保证模块何时会执行。研究以下代码：

```
<!-- no guarantee which one of these will execute first -->
<script type="module" async src="module1.js"></script>
<script type="module" async src="module2.js"></script>
```

此例中两个模块文件被异步加载了。仅查看代码是不可能判断出哪个模块会被先执行。若 `module1.js` 首先结束下载（包括它的所有导入资源），那么它就会首先执行。而对于 `module2.js` 来说也是一样。

将模块作为 **Worker** 加载

诸如 Web Worker 与 Service Worker 之类的 worker ，会在网页上下文外部执行 JS 代码。创建一个新的 worker 调用，也就会创建 `Worker` （或其他 worker 类）的一个实例，并会向其传入 JS 文件的位置。其默认的加载机制是将文件当作脚本来下载，例如：

```
// 用脚本方式加载 script.js
let worker = new Worker("script.js");
```

为了支持模块加载， HTML 标准的开发者为这些 worker 构造器添加了第二个参数，此参数是一个有 `type` 属性的对象，该属性的默认值是 `"script"` 。你也可以将 `type` 设置为 `"module"` 以便加载模块文件：

```
// 用模块方式加载 module.js
let worker = new Worker("module.js", { type: "module" });
```

此例通过传递 `type` 属性值为 `"module"` 的第二个参数，将 `module.js` 作为模块而不是脚本进行了加载（ `type` 属性也就是模拟了 `<script>` 标签在模块与脚本之间的 `type` 区别）。这第二个参数在浏览器中的所有的 worker 类型中都得到了支持。

worker 模块通常与 worker 脚本一致，但存在两点例外。首先， worker 脚本被限制只能从同源的网页进行加载，而 worker 模块可以不受此限制。尽管 worker 模块具有相同的默认限制，但当响应头中包含恰当的跨域资源共享（ Cross-Origin Resource Sharing ， CORS ）时，就允许跨域加载文件。其次， worker 脚本可以使用 `self.importScripts()` 方法来将额外脚本引入 worker ，而 worker 模块上的 `self.importScripts()` 却总会失败，因为应当换用 `import` 。

浏览器模块说明符方案

本章至今的所有范例都使用了相对的模块说明符，例如 `"./example.js"` 。浏览器要求模块说明符应当为下列格式之一：

- 以 `/` 为起始，表示从根目录开始解析；
- 以 `./` 为起始，表示从当前目录开始解析；
- 以 `../` 为起始，表示从父级目录开始解析；
- URL 格式。

例如，假设你拥有一个位于 `https://www.example.com/modules/module.js` 的模块文件，包含了以下代码：

```
// 从 https://www.example.com/modules/example1.js 导入
import { first } from "./example1.js";

// 从 from https://www.example.com/example2.js 导入
import { second } from "../example2.js";

// 从 from https://www.example.com/example3.js 导入
import { third } from "/example3.js";

// 从 from https://www2.example.com/example4.js 导入
import { fourth } from "https://www2.example.com/example4.js";
```

此例中每一个模块说明符在浏览器中使用时都是有效的，包括最后一行的完整 URL （你无须确保 `ww2.example.com` 已经正确配置了它的 CORS 响应头来允许跨域加载，这会影响是否能跨域加载，却不会影响语法的有效性）。只有这些才是浏览器默认情况能使用的模块说明符格式，不过未完成的模块加载器规范将会提供对其他格式的支持。这意味着某些看似正常的模块说明符实际上在浏览器中是无效的，并且会导致错误，正如：

```
// 无效：没有以 / 、 ./ 或 ../ 开始
import { first } from "example.js";

// 无效：没有以 / 、 ./ 或 ../ 开始
import { second } from "example/index.js";
```

此处的模块说明符都不能被浏览器加载。这两个模块说明符都缺失了正确的起始字符，使用的是无效格式，尽管在 `<script>` 标签中作为 `src` 来使用是有效的。这是在 `<script>` 与 `import` 之间有意制造的行为差异。

总结

ES6 为 JS 语言添加了模块，作为打包与封装功能的方式。模块的行为异于脚本，它们不会用自身顶级作用域的变量、函数或类去修改全局作用域，而模块的 `this` 值为 `undefined` 。为了实现这些行为，模块在被加载时使用了一种不同的方式。

你必须将模块中需要向外提供的任何功能都导出，变量、函数与类都可以，并且每个模块允许存在一个默认导出。在导出之后，另一个模块就能导入该模块所导出的一个或多个名称了。这些导入的名称就像是被 `const` 所定义的，会被当作块级绑定，并且不允在同一模块内重复声明。

如果模块只是要操纵全局作用域，那么无须导出任何值。你实际上可以导入这样一个模块，而不会在当前模块作用域中引入任何绑定。

由于模块必须用与脚本不同的方式运行，浏览器就引入了 `<script type="module">` ，以表示资源文件或内联代码需要作为模块来执行。使用 `<script type="module">` 加载的模块文件会默认应用 `defer` 属性。一旦包含模块的页面文档完全被解析，模块就会按照它们在文档中的出现顺序依次执行。

附录A：较小的改进

除了本书已涵盖的主要变化之外，ES6 还做出了一些虽小但仍然有助于改进 JS 的变更，包括：使整型更易用、新增计算方法、对 Unicode 标识符的细微调整，以及规范化 `__proto__` 属性。我会在本附录中描述所有这些内容。

处理整型

JS 使用 IEEE 754 编码系统来表示整型与浮点型，多年以来这引发了很多混乱。虽然这门语言煞费苦心地确保开发者不需要关心数值的编码细节，但问题仍然会时不时涌现出来。 ES6 力图解决这方面的问题，让整型变得更易识别、更易处理。

识别整型

首先， ES6 新增了 `Number.isInteger()` 方法，用于判断一个值是否能在 JS 中表示整型。虽然 JS 使用了 IEEE 754 来同时表示浮点型与整型这两种数值，但它们的存储方式仍有差异。 `Number.isInteger()` 方法利用了这个差异，当使用一个值来调用此方法时， JS 引擎会查看该值的底层表示以判断它是不是一个整型。这意味着看起来像浮点型的数值实际上可能被存储为整型，此时 `Number.isInteger()` 便会返回 `true` 。例如：

```
console.log(Number.isInteger(25));      // true
console.log(Number.isInteger(25.0));    // true
console.log(Number.isInteger(25.1));    // false
```

在此代码中，向 `Number.isInteger()` 传入 `25` 和 `25.0` 都会返回 `true` ，尽管后者看起来是浮点型。在 JS 中，单纯添加一个小数点并不会让数字自动变为浮点型。由于 `25.0` 实际上就是 `25` ，它就被存储为整型；而数值 `25.1` 则会被存储为浮点型，因为它拥有小数部分。

安全的整型

IEEE 754 只能精确表示 -2^{53} 与 2^{53} 之间的整型数，在该“安全”范围之外，多个不同的数值就有可能对应同一个二进制表示。这意味着 JS 只能在 IEEE 754 的精确范围内保证对整型数的安全表示。例如，研究以下例子：

```
console.log(Math.pow(2, 53));       // 9007199254740992
console.log(Math.pow(2, 53) + 1);   // 9007199254740992
```

此例并不包含拼写错误，然而两个不同的数值却被表示成了同一个 JS 整型数。当数值超出安全范围越远，此效果就越加明显。

ES6 引进了 `Number.isSafeInteger()` 以便更好识别该语言所能精确表示的整型；同时新增的还有 `Number.MAX_SAFE_INTEGER` 与 `Number.MIN_SAFE_INTEGER` 属性，分别用于表示整型数的上下边界。 `Number.isSafeInteger()` 方法能确认一个值是整型、并且它落在安全范围之内，正如此例：

```
var inside = Number.MAX_SAFE_INTEGER,
    outside = inside + 1;

console.log(Number.isInteger(inside));          // true
console.log(Number.isSafeInteger(inside));      // true

console.log(Number.isInteger(outside));         // true
console.log(Number.isSafeInteger(outside));     // false
```

数值 `inside` 是最大的安全整型数，因此用它去调用 `Number.isInteger()` 与 `Number.isSafeInteger()` 都会返回 `true` 。数值 `outside` 则是第一个可疑的整型值，尽管它依然是个整型数，但仍被认为是不安全的。

多数情况下，你只想用安全的整型数在 JS 中去进行整型运算或比较，因此使用 `Number.isSafeInteger()` 作为输入验证的一部分便是个好主意。

新的数学方法

ES6 的游戏与图形的新重点引导它将类型化数组（ typed array ）引入了 JS ，同时也让它意识到 JS 引擎应当更有效率地进行许多数学计算。但诸如 asm.js （工作在 JS 的一个子集上以提高效率）之类的优化策略，都需要更多的信息以便尽可能快地进行计算。例如，知道数值是被作为 32 位整型还是 64 位浮点型来处理，对基于硬件的操作来说是非常重要的，而这要比基于软件的操作快得多。

因此， ES6 给 `Math` 对象新增了几个方法来提高通用数学计算的速度，而提高通用计算速度也能让图形程序之类的需要进行密集计算的应用提高总体速度。下列就是这些新方法：

- `Math.acosh(x)` ：返回 x 的反双曲余弦值；
- `Math.asinh(x)` ：返回 x 的反双曲正弦值；

- `Math.atanh(x)` ：返回 x 的反双曲正切值；
- `Math.cbrt(x)` ：返回 x 的立方根；
- `Math.clz32(x)` ：返回 x 的 32 位整型二进制表达形式起始处 0 的个数；
- `Math.cosh(x)` ：返回 x 的双曲余弦值；
- `Math.expm1(x)` ：返回 e^x - 1 的值；
- `Math.fround(x)` ：返回最接近 x 的单精度浮点数；
- `Math.hypot(...values)` ：返回参数平方和的平方根；
- `Math.imul(x, y)` ：返回两个参数真正的 32 位乘法运算结果；
- `Math.log1p(x)` ：返回 1 + x 的自然对数；
- `Math.log10(x)` ：返回 x 的常用对数（即以 10 为底）；
- `Math.log2(x)` ：返回 x 的二进制对数（即以 2 为底）；
- `Math.sign(x)` ： x 为负数时返回 -1 ， +0 与 -0 返回 0 ，正数则返回 1 ；
- `Math.sinh(x)` ：返回 x 的双曲正弦值；
- `Math.tanh(x)` ：返回 x 的双曲正切值；
- `Math.trunc(x)` ：移除浮点型数值小数点后的数字，以返回一个整型值。

解释每个新方法以及它的细节已经超出了本书的范围。不过若你的应用需要合理地进行通用计算，在自行实现之前请先检查是否已有对应的 `Math` 新方法。

Unicode 标识符

ES6 提供了比之前版本更好的 Unicode 支持，同时也修改了能被用于标识符的字符范围。在 ES5 中已经能在标识符里使用 Unicode 转义序列，例如：

```
// 在 ES5 与 ES6 中都有效
var \u0061 = "abc";

console.log(\u0061);     // "abc"

// 等价于：
console.log(a);          // "abc"
```

在此例的 `var` 语句之后，你用 `\u0061` 或 `a` 都能访问这个变量。在 ES6 中，你还能在标识符里使用 Unicode 码点转义序列，就像这样：

```
// 在 ES5 与 ES6 中都有效
var \u{61} = "abc";

console.log(\u{61});      // "abc"

// 等价于：
console.log(a);          // "abc"
```

本例只是将 `\u0061` 替换为它的码点等价形式，这么做的效果实际上与上个例子完全相同。

另外， ES6 还将 Unicode 标准附录 31 中的字符正式指定为有效的标识符（该附录详见 Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax），其规则如下：

1. 第一个字符必须是 `$` 、 `_` ，或任何属于 `ID_start` 核心衍生属性的 Unicode 符号；
2. 之后的字符必须为 `$` 、 `_` 、 `\u200c` （零宽不连字）、 `\u200d` （零宽连字），或任何属于 `ID_Continue` 核心衍生属性的 Unicode 符号。

`ID_Start` 与 `ID_Continue` 核心衍生属性由 Unicode 标识符与模式语法（即上述附录）定义，提供了一种方法以识别能被用于标识符（如变量与域名）的合适符号。该规范并未针对 JS 。

规范化的 `__proto__` 属性

在 ES5 规范完成之前，几个 JS 引擎就已经实现了一个称为 `__proto__` 的自定义属性，能用它来获取并设置 `[[Prototype]]` 属性。实际上， `__proto__` 就是 `Object.getPrototypeOf()` 与 `Object.setPrototypeOf()` 方法的早期先驱。期望所有的 JS 引擎都移除这个属性是不现实的，因为有些流行的 JS 代码库已经利用了该属性，因此 ES6 也将该属性的行为标准化了，但在 ECMA-262 附录 B 中该规范也附带了以下警告：

> 这些特性并不被认为是 ES 语言的核心部分，程序员在书写新的 ES 代码时，不应使用它、或假定这些特性存在。 ES 的实现方案并不鼓励实现这些特性，除非该实现已是 web 浏览器的一部分、或者被用于在浏览器中运行遗留代码。

ES 规范更推荐使用 `Object.getPrototypeOf()` 与 `Object.setPrototypeOf()` 方法，因为 `__proto__` 具有如下特征：

1. `__proto__` 在对象字面量中只能指定一次，指定多个 `__proto__` 将会抛出错误。这也是对象字面量属性中唯一受此限制的属性。
2. 对象字面量中可计算形式的 `["__proto__"]` 表现得就像是常规属性，并不会设置或返回当前对象的原型。对于字面量属性来说，可计算形式与非计算形式一般是等价的，只有 `__proto__` 例外。

> 译注：这代表以下两种写法并不等价——
>
> ```
> let a = {
>    ["__proto__"]: Number
> };
> ```
>
> 以及
>
> ```
> let b = {
>    __proto__: Number
> };
> ```
>
> 后者可以将 `b` 的原型设置为 `Number` ，而前者对 `a` 的原型没有造成任何影响。
>
> 在上述代码执行后，再执行 `b.isInteger(15.2)` 会返回 `false` ，而执行 `a.isInteger(15.2)` 则会报错，因为 `Number` 并没有正确绑定到 `a` 的原型上，只是作为一个普通属性存在。

你应当规避 `__proto__` 属性，不过规范文档定义它的方式却很有意思。在 ES6 引擎中， `Object.prototype.__proto__` 被定义为一个访问器属性，其 `get` 方法会调用 `Object.getPrototypeOf()` ，而 `set` 方法则会调用 `Object.setPrototypeOf()` 。这样在使用 `__proto__` 与使用 `Object.getPrototypeOf()` / `Object.setPrototypeOf()` 之间就几乎没有真正区别，唯一例外是 `__proto__` 能在对象字面量中直接使用，用于设置对象的原型。以下是使用它的范例：

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

let dog = {
    getGreeting() {
        return "Woof";
    }
};

// 原型设为 person
let friend = {
    __proto__: person
};
console.log(friend.getGreeting());                      // "Hello"
console.log(Object.getPrototypeOf(friend) === person);  // true
console.log(friend.__proto__ === person);               // true

// 将 prototype 改为 dog
friend.__proto__ = dog;
console.log(friend.getGreeting());                      // "Woof"
console.log(friend.__proto__ === dog);                  // true
console.log(Object.getPrototypeOf(friend) === dog);     // true
```

此例未调用 `Object.create()` 来创建 `friend` 对象，而是创建了一个标准的对象字面量，并将一个值（ `person` ）赋给了 `__proto__` 属性。而另一方面，当使用 `Object.create()` 方式时，你需要为对象的任意附加属性指定完整的属性描述符。

附录B：理解ES7(ES2016)

ES6 的开发消耗了大约四年时间，此后 TC-39 小组决定不再接受如此长的开发过程，他们改用年度周期来发布语言的新特性，以确保语言特性能够尽快发展。

更频繁的发布意味着每个新的 ES 版本拥有的新特性会比 ES6 少得多。为了标示这种变化，新规范的版本号不再重点描述版本数字，而改为指明规范发布的年份。因此 ES6 也被称为 ES2015 ，而 ES7 也正式被称为 ES2016 。 TC-39 小组预计将来所有的 ES 版本都会使用这个以年份为基础的命名系统。

ES2016 在 2016 年 3 月定稿，并且只向语言添加了三项内容：一个新的数学运算符、一个新的数组方法，以及一种新的语法错误。这些全都包含在本附录中。

- 幂运算符
  - 运算顺序
  - 操作数的限制
- Array.prototype.includes() 方法
  - 如何使用 Array.prototype.includes()
  - 值比较
- 函数作用域严格模式的改动

幂运算符

ES2016 对 JS 语法引入的唯一改进就是幂运算符（ **exponentiation operator** ），此数学运算能将指数运用到底数上。 虽然已经有 `Math.pow()` 方法可用于幂运算，然而 JS 也是在此方面只能调用方法而不能使用运算符的少数语言之一，并且一些开发者认为运算符可读性更强、更易于检查。

幂运算符是两个星号（ `**` ），其左侧是底数，右侧则是指数，例如：

```
let result = 5 ** 2;

console.log(result);                        // 25
console.log(result === Math.pow(5, 2));     // true
```

此例计算了 5^2 ，结果为 25 。你也可以使用 `Math.pow()` 来获取相同结果。

运算顺序

幂运算符的优先级在 JS 的二元运算符中是最高的（但低于一元运算符）。这意味着在复合运算中它会被优先运用，正如下例：

```
let result = 2 * 5 ** 2;
console.log(result);        // 50
```

5^2 的计算会首先发生，其结果值随后会与 2 相乘来产生最终的结果： 50 。

操作数的限制

与其他运算符不同，幂运算符有一个稍显独特的限制：运算符左侧不能是除了 `++` 或 `--` 之外的任意一元表达式，例如，下面的语法是无效的：

```
// 语法错误
let result = -5 ** 2;
```

本例中的 `-5` 是个语法错误，因为运算的顺序是有二义性的。 `-` 应当运用到 `5` 上面，还是运用到 `5 ** 2` 的结果上面呢？禁止左侧的一元表达式消除了这个歧义。为了清晰地说明意图，需要给 `-5` 或 `5 ** 2` 添加圆括号，如下所示：

```
// 没问题
let result1 = -(5 ** 2);    // 等于 -25

// 也没问题
let result2 = (-5) ** 2;    // 等于 25
```

若你为幂运算表达式包裹括号，那么 `-` 会作用在这整个表达式上；而若你为 `-5` 包裹括号，则清楚表示想要获取 -5 的平方。

在幂运算符的左侧表达式使用的是 `++` 或 `--` 时，就无需使用括号，因为这两个运算符在操作数上的行为都被清晰定义了： `++` 或 `--` 作为前缀会在其他任意运算发生之前修改操作数，而作为后缀则会在整个表达式计算完毕后才修改操作数。在幂运算符左侧的这两种用法都是安全的，正如下面代码所示：

```
let num1 = 2,
    num2 = 2;

console.log(++num1 ** 2);       // 9
console.log(num1);              // 3

console.log(num2-- ** 2);       // 4
console.log(num2);              // 1
```

此例中的 `num1` 在幂运算符被运用之前就已被递增，因此 `num1` 会变为 3 ，并且幂运算的结果为 9 。而对于 `num2` 来说，在参与幂运算时它的值仍然保持为 2 ，在运算之后才被递减到 1 。

Array.prototype.includes() 方法

你或许会记得 ES6 新增了 `String.prototype.includes()` 用于检查某个子字符串是否存在于指定字符串中。 ES6 起初也想引入一个 `Array.prototype.includes()` 方法，让使用类似方式处理字符串与数组的倾向能得以保持。但此方法的规范未能赶上 ES6 的最后期限，于是最终它就进入了 ES2016 。

如何使用 Array.prototype.includes()

`Array.prototype.includes()` 方法接受两个参数：需要搜索的值、可选的搜索起始位置索引。当提供了第二个参数时， `includes()` 会从该位置开始尝试匹配（默认的起始位置为 0 ）。若在数组中找到了该值，返回 `true` ；否则返回 `false` 。例如：

```
let values = [1, 2, 3];

console.log(values.includes(1));        // true
console.log(values.includes(0));        // false

// 从索引 2 开始搜索
console.log(values.includes(1, 2));     // false
```

此处使用 `1` 去调用 `values.includes()` 返回了 `true` ，而使用 `0` 则会返回 `false` ，因为 `0` 并不在此数组中。当使用了第二个参数让搜索从索引位置 2 （该位置的元素值是 `3` ）开始进行时， `values.includes()` 方法返回了 `false` ，因为数值 `1` 并不存在于位置 2 与数组末端之间。

值比较

`includes()` 方法使用 `===` 运算符来进行值比较，仅有一个例外： `NaN` 被认为与自身相等，尽管 `NaN === NaN` 的计算结果为 `false` 。这与 `indexOf()` 方法的行为不同，后者在比较时严格使用了 `===` 运算符。为了明白其中的差异，研究如下代码：

```
let values = [1, NaN, 2];

console.log(values.indexOf(NaN));       // -1
console.log(values.includes(NaN));      // true
```

`values.indexOf()` 方法为 `NaN` 返回了 `-1` ，尽管 `NaN` 实际上被包含在 `values` 数组中。另一方面，由于使用不同的比较运算符， `values.includes()` 方法则为 `NaN` 返回了 `true` 。

> 若只想检查某个值是否存在于数组中，而不想知道它的位置，我推荐使用 `includes()` ，这是由于 `includes()` 与 `indexOf()` 方法对于 `NaN` 的处理不同。而若确实想知道某个值在数组中的位置，那么就必须使用 `indexOf()` 方法。

在实现中的另一个怪异点是 `+0` 和 `-0` 被认为是相等的。在这个方面， `indexOf()` 与 `includes()` 行为一致：

```
let values = [1, +0, 2];

console.log(values.indexOf(-0));        // 1
console.log(values.includes(-0));       // true
```

此处的 `indexOf()` 与 `includes()` 都在传入 `-0` 的时候找到了 `+0` ，因为它们认为这两个值是相等的。注意这与 `Object.is()` 方法的行为有差异，后者认为 `+0` 与 `-0` 是不同的值。

函数作用域严格模式的改动

当严格模式在 ES5 中被引入时， JS 语言要比后来的 ES6 简单得多。尽管如此， ES6 仍然允许你使用 `"use strict"` 指令来指定严格模式，可以在全局作用域上，让所有代码运行在严格模式下；或用在函数作用域内，只有该函数会运行在严格模式下。后一种方式最终在 ES6 中成为了一个问题，这是由于参数可以用更加复杂的方式来定义，特别是在带有解构或默认值的情况下。为了理解这个问题，研究如下代码：

```
function doSomething(first = this) {
    "use strict";

    return first;
}
```

此处的具名参数 `first` 被赋予了一个默认值 `this` ，而你预期 `first` 的值会是什么？ ES6 规范指示 JS 引擎此时要用严格模式来处理参数，因此 `this` 的值应当等于 `undefined` 。然而，当函数内部指定了 `"use strict"` 时， 要将参数也限制在严格模式中有相当的困难，因为参数默认值也可能是个函数。这种困难情况导致大部分 JS 引擎都没有实现这个特性（因此 `this` 的值可能等于全局对象）。

由于该实现的难度， ES2016 规定如果函数的参数被进行解构或是拥有默认值，则在该函数内部使用 `"use strict"` 指令将是违法的。当函数体内出现了 `"use strict"` 时，该函数只允许使用简单参数列表（也就是所包含的参数没有进行解构，也没有默认值）。下面有一些示例：

```
// 没有问题，使用了简单参数列表
function okay(first, second) {
    "use strict";

    return first;
}

// 语法错误
function notOkay1(first, second=first) {
    "use strict";

    return first;
}

// 语法错误
function notOkay2({ first, second }) {
    "use strict";

    return first;
}
```

你依旧可以在只有简单参数列表的函数内使用 `"use strict"` ，这也是 `okay()` 函数如预期正常工作的原因（就像在 ES5 中一样）。 `notOkay1()` 函数则会有语法错误，因为它使用了参数的默认值。类似的， `notOkay2()` 函数同样有语法错误，因为它使用了解构参数。

总的来说，这项改动既解决了 JS 开发者的困惑，又消除了 JS 引擎的一个实现难题。

- 2017-09-01
  - 第十一章 Promise与异步编程
    - 使用 Promise.reject()
    - 异步任务运行
- 2017-08-20
  - 总体修改说明
  - 第三章 函数
    - 剩余参数
  - 第四章 扩展的对象功能
    - 使用 super 引用的简单原型访问
  - 第五章 解构：更方便的数据访问
    - 参数解构的默认值
  - 第六章 符号与符号属性
    - 符号值的转换
    - 使用知名符号暴露内部方法
  - 第七章 Set与Map
    - 使用 Weak Map
    - 对象的私有数据
  - 第八章 迭代器与生成器
    - 字符串的迭代器
    - 传递参数给迭代器
  - 第十章 增强的数组功能
    - 在可迭代对象上使用
    - 类型化数组
  - 第十二章 代理与反射接口
    - 使用 get 陷阱函数进行对象外形验证
    - 将代理作为类的原型
  - 第十三章 用模块封装代码
    - 浏览器模块说明符方案
    - 总结
- 2017-08-02
  - 第三章 函数
    - 参数默认值表达式
    - 名称属性的特殊情况
    - 明确函数的双重用途
    - new.target 元属性
- 2017-07-04
  - 第二章 字符串与正则表达式
    - 识别子字符串的方法
- 2017-03-26

  - 第二章 字符串与正则表达式
    - 识别子字符串的方法
  - 第三章 函数
    - `new.target` 元属性
    - 没有 this 绑定
  - 第六章 符号与符号属性
    - 使用符号值
    - Symbol.match 、 Symbol.replace 、 Symbol.search 与 Symbol.split
  - 第七章 Set与Map
    - 创建 Set 并添加项目
    - Set 类型之间的关键差异
  - 第八章 迭代器与生成器
    - 生成器委托
    - 解构与 `for-of` 循环
  - 第九章 JS的类
    - 基本的类表达式
    - 继承内置对象
  - 第十二章 代理与反射接口
    - 起始第二段
    - ownKeys 陷阱函数

2017-09-01

第十一章 Promise与异步编程

使用 Promise.reject()

> 若你传递一个 Promise 给 `Promise.resolve()` 或 `Promise.reject()` 方法，该 Promise 会不作修改原样返回。

修改为：

> 若你传递一个 Promise 给 `Promise.resolve()` ，该 Promise 会不作修改原样返回。

并删除以下译注：

> 译注：经过测试，在几大浏览器中都存在与上一句话不符的情况。
>
> 1. 若传入的 Promise 为进行态，则 `Promise.resolve()` 调用会将该 Promise 原样返回。此后，若决议原 Promise ，在 `then()` 中可以接收到原例中的参数 `42` ；而若拒绝原 Promise ，则在 `catch()` 中可以接收到参数 `42` 。但 `Promise.reject()` 调用则会对原先的 Promise 重新进行包装，对其使用 `catch()` 可以捕捉到错误，处理函数中的 `value` 参数不会是数值 `42` ，而是原先处于进行态的 Promise 。
> 2. 若传入的 Promise 为完成态，则 `Promise.resolve()` 调用会将该 Promise 原样返回，在 `then()` 中可以接收到原例中的参数 `42` 。但 `Promise.reject()` 调用则会对原先的 Promise 重新进行包装，对其使用 `catch()` 可以捕捉到错误，处理函数中的 `value` 参数不会是数值 `42` ，而是原先处于完成态的 Promise 。
> 3. 若传入的 Promise 为拒绝态，则 `Promise.reject()` 调用会将该 Promise 原样返回，在 `catch()` 中可以接收到参数 `42` 。但 `Promise.resolve()` 调用则会对原先的 Promise 重新进行包装，对其使用 `then()` 可以进行完成处理，处理函数中的 `value` 参数不是 `42` ，而是原先处于拒绝态的 Promise 。也就是说此时的情况与上一种情况相反。
>
> 总结：对进行态或完成态的 Promise 使用 `Promise.resolve()` 没问题，会返回原 Promise ；对拒绝态的 Promise 使用 `Promise.reject()` 也没问题。而除此之外的情况全都会在原 Promise 上包装出一个新的 Promise 。

在目前的浏览器中测试结果为： `Promise.resolve()` 会将传入的 Promise 类型的参数直接返回，无论该 Promise 处于何种状态。而 `Promise.resolve()` 则会对传入的任意对象参数进行 Promise 包装，返回一个新的 Promise 对象，无论传入参数是否为 Promise 类型，也不论参数是否为已处于拒绝态的 Promise 。这样的行为也符合了 ES6 的规范，其中 `Promise.resolve()` 会根据对象参数的 `constructor` 属性来判断它是否为一个 Promise 对象。

上述附注已不符合实际情况，因此予以删除。原文唯一问题是对于 `Promise.reject()` 方法的使用。

异步任务运行

删除以下译注：

> 译注：此处的
>
> > `Promise.resolve()` 在被传入任意 Promise 时只会直接将其传递回来
>
> 表述不够准确，详情参见前面的译注。

理由同上。

2017-08-20

总体修改说明

对全部译文重新进行了检查，修改了 700 多处。不过绝大部分仅仅是修改了表述方式，包括调整语序、修改关联词、减少括号的使用等。此外：

1. 原文中对其他章节的引用，许多地方提到的章节序号都有错，都进行了修正，不一一列出。
2. 根据正式中文版图书，对少数名词的翻译进行了调整，包括：
    - 将“需计算属性名”更改为“可计算属性名”。
    - 将“Object.assign() 方法”一节中提到的“供应者”更改为“源对象”。
    - “一级公民”与“一级函数”改为“一等公民”与“一等函数”，两种译法都有，但后者貌似更常见一些。
    - Promise 部分，将 `pending` 、 `unsettled` 与 `settled` 分别改译为“进行中”、“未处理”、“已处理”。
3. 除此之外的错误都列在了后面，其中有些是原作的问题，有些则是翻译错误。

第三章 函数

剩余参数

> 函数的 `length` 属性用于指示具名参数的数量，而剩余参数对其毫无影响。此例中 `pick()` 函数的 `length` 属性值是 1 ，因为只有 `object` 参数被用于计算该值。

这段话后添加一段译注：

> 译注：这种说法并不严谨。若函数使用了默认参数，则 **`length`** 属性不包含使用默认值的参数，并且它只能指示出第一个默认参数之前的具名参数数量。例如对于 **`function example(first, second = 'woo', third) {}`** 函数声明来说， **`length`** 的值是 **`1`** 而非 **`2`** ，尽管这里有两个无默认值的具名参数。

第四章 扩展的对象功能

使用 super 引用的简单原型访问

> 此例使用了一个函数作为具名方法

修改为：

> 此例使用了一个匿名函数作为属性

原文的 named 用于修饰属性，而不是函数，该函数实际上是个匿名函数。

第五章 解构：更方便的数据访问

参数解构的默认值

在这一小节最后部分添加附注：

> 上面的参数解构只有一个缺点，也就是当传入参数值为 `null` 时会引发程序异常。只有 `null` 与 `undefined` 是无法被解构的，而传入 `undefined` 会触发默认参数的使用条件，从而避免了异常；但传入 `null` 就不会有这么幸运了，它既不会触发默认参数，也不能被解构，从而导致异常。

第六章 符号与符号属性

符号值的转换

> 但若你想直接将符号转换为字符串，则会引发错误

修改为：

> 但若你想将符号与字符串直接拼接，则会引发错误

使用知名符号暴露内部方法

> 它只是改变了规范所描述的对象特征

改为：

> 它只是改变了规范描述对象的方式

第七章 Set与Map

使用 Weak Map

“非空对象”修改为“非 null 的对象”，后面“Weak Map 的初始化”小节也要作此修改。

“空对象”在 js 中一般指的是 `{}` ，这与 `null` 并不一样。

对象的私有数据

> 此例用 IIFE 包裹了 `Person` 的定义，其中含有两个私有属性

修改为：

> 此例用 IIFE 包裹了 `Person` 的定义，其中含有两个私有变量

第八章 迭代器与生成器

字符串的迭代器

> 因此它不能被用于正确访问双字节的字符

修改为：

> 因此它不能被用于正确访问双码元的字符

属于原作的错误。前面曾经介绍过，编码单元本身就是两个字节（16位）的， JS 从前就可以处理一个码元（双字节）的字符，例如各种中文字符。旧版 JS 所不能处理的是双编码单元（一共32位，即四个字节）的字符，例如后面提到的那个日文字符。

同理——

> 由于双字节字符被当作两个分离的码元来对待

修改为：

> 由于四字节字符被当作两个分离的码元来对待

传递参数给迭代器

关于“首次调用 next() 时传递参数是无效”这一问题，将原文的翻译与译者个人的理解对调了一下位置，后者放到附注部分。

第十章 增强的数组功能

在可迭代对象上使用

> 如果一个对象既是类数组对象，又是可迭代对象，那么迭代器就会使用 `Array.from()` 方法来决定需要转换的值。

修改为：

> 如果一个对象既是类数组对象，又是可迭代对象，那么 `Array.from()` 方法就会使用迭代器来决定需要转换的值。

类型化数组

> 而不像名称暗示的那样，能处理所有类型

修改为：

> 正如名称所示，不是所有类型

第十二章 代理与反射接口

使用 get 陷阱函数进行对象外形验证

> `receiver` 并没有使用 `trapTarget` ，而是用了 `in`

修改为：

> 使用 `receiver` 而非 `trapTarget` 去配合 `in`

将代理作为类的原型

> `shape` 的原型是 `Shape.prototype` ，它并不是一个代理。

原作笔误，应修改为：

> `shape` 的原型是 `Square.prototype` ，它并不是一个代理。

第十三章 用模块封装代码

浏览器模块说明符方案

> 你无须确保 `ww2.example.com` 已经正确配置了它的 CORS 响应头来允许跨域加载，这会影响是否能跨域加载，却不会影响语法的有效性

修改为：

> 你必须确保 `ww2.example.com` 已经正确配置了它的 CORS 响应头来允许跨域加载

总结

> 这些导入的名称就像是被 `let` 所定义的

修改为：

> 这些导入的名称就像是被 `const` 所定义的

原作在此处的描述不符合之前的文字，并且与实际情况也有偏差。

2017-08-02

第三章 函数

参数默认值表达式

> 而在 `getValue()` 的函数声明初次被解析时并不会进行调用。这意味着 `getValue()` 函数若被写为可变的，则它有可能会返回可变的值

修改为：

> 而在 `add()` 的函数声明初次被解析时并不会进行调用。这意味着 `getValue()` 函数若被写为可变的，则它返回的值有可能也会变化

名称属性的特殊情况

> 而使用 `Function` 构造器创建的函数，其名称属性则会有 `"anonymous"` 前缀

修改为：

> 而使用 `Function` 构造器创建的函数，其名称属性为 `"anonymous"`

明确函数的双重用途

> 当函数未使用 `new` 进行调用时， `[[call]]` 方法会被执行

修改为：

> 当函数未使用 `new` 进行调用时， `[[Call]]` 方法会被执行

首字母需要大写。

new.target 元属性

> 元属性指的是“非对象”（例如 `new` ）上的一个属性，并提供关联到它的目标的附加信息。当函数的 `[[Construct]]` 方法被调用时， `new.target` 会被填入 `new` 运算符的作用目标，该目标通常是新创建的对象实例的构造器，并且会成为函数体内部的 `this` 值。

修改为：

> 元属性指的是“非对象”（例如 `new` 运算符）上的属性，并提供与之关联的目标的附加信息。当函数的 `[[Construct]]` 方法被调用时， `new` 运算符的作用目标会填入 `new.target` 元属性，此时函数体内部的 `this` 值是新创建的对象实例，而 `new.target` 的值正是该实例的构造器（构造函数）。

2017-07-04

第二章 字符串与正则表达式

识别子字符串的方法

此处原文对于 `endsWith()` 方法第二个参数的描述有误。

> 当提供了第二个参数时， `includes()` 与 `startsWith()` 方法会从该索引位置开始尝试匹配；而 `endsWith()` 方法会将字符串长度减去该参数，以此为起点开始尝试匹配。当第二个参数未提供时， `includes()` 与 `startsWith()` 方法会从字符串起始处开始查找，而 `endsWith()` 方法则从尾部开始。

修改为：

> 当提供了第二个参数时， `includes()` 与 `startsWith()` 方法会从该索引位置开始尝试匹配；而 `endsWith()` 方法则会将该位置减去需搜索文本的长度，在目标位置尝试匹配。当第二个参数未提供时， `includes()` 与 `startsWith()` 方法会从字符串起始处开始查找，而 `endsWith()` 方法则从尾部开始（事实上是在原字符串长度减去需搜索文本长度的位置上进行正向查找）。

此外，

> 调用 `msg.endsWith("o", 8)` 也从位置 4 开始搜索，因为参数 `8` 会从字符串的长度值（ 12 ）中被减去；

修改为：

> 调用 `msg.endsWith("o", 8)` 会在位置 7 进行匹配尝试，因为需要将参数 `8` 减去字符串 `"o"` 的长度 `1` ；

2017-03-26

第二章 字符串与正则表达式

识别子字符串的方法

> 前三次调用没有使用第二个参数

修改为：

> 前六次调用没有使用第二个参数

第三章 函数

`new.target` 元属性

> 译注：原文此段代码有误。
>
> ```
> if (new.target === Person) {
> ```
>
> 这一行原先写为：
>
> ```
> if (typeof new.target === Person) {
> ```
>
> 原先的写法是有问题的，不能正确发挥作用，它会在 `new Person("Nicholas")` 这行就抛出错误。

删掉这一段译注。由于向原作者提交 issues 并已经被确认，原文已修改，此段译注就没有必要保留了。

没有 this 绑定

> 否则， `this` 值就会是 `undefined` 。

改为：

> 否则， `this` 值就会是全局对象（在浏览器中是 `window` ，在 nodejs 中是 `global` ）。

第六章 符号与符号属性

使用符号值

> 这个例子首先使用对象的“需计算字面量属性”方式创建了一个符号类型的属性 `firstName` ，该属性默认不可枚举，此行为与非符号类型的“需计算字面量属性名”正相反。

修正为：

> 这个例子首先使用对象的“需计算字面量属性”方式创建了一个符号类型的属性 `firstName` ，该属性使用 getOwnPropertyDescriptor 查看时显示为可枚举（ `enumerable: true` ），但无法用 for-in 循环遍历，也不会显示在 `Object.keys()` 的结果中。

Symbol.match 、 Symbol.replace 、 Symbol.search 与 Symbol.split

> 使得开发者无法将自定义对象模拟成正则表达式

稍微添加点附注，让意思更明确：

> 使得开发者无法将自定义对象模拟成正则表达式（并将它们传递给字符串的这些方法）

此外：

> 这 4 个符号表示可以将正则表达式作为对应方法的第一个参数传入

修改为：

> 这 4 个符号表示可以将正则表达式作为字符串对应方法的第一个参数传入

范例代码：

```
    [Symbol.match]: function(value) {
        return value.length === 10 ? [value.substring(0, 10)] : null;
    },
    [Symbol.replace]: function(value, replacement) {
        return value.length === 10 ?
            replacement + value.substring(10) : value;
    },
```

修改为：

```
    [Symbol.match]: function(value) {
        return value.length === 10 ? [value] : null;
    },
    [Symbol.replace]: function(value, replacement) {
        return value.length === 10 ? replacement : value;
    },
```

第七章 Set与Map

创建 Set 并添加项目

> （在 Set 内部的比较使用了第四章讨论过的 `Object.is()` 方法，来判断两个值是否相等）

添加文字，改为：

> （在 Set 内部的比较使用了第四章讨论过的 `Object.is()` 方法，来判断两个值是否相等，唯一的例外是 +0 与 -0 在 Set 中被判断为是相等的）

Set 类型之间的关键差异

> 1. 对于 `WeakSet` 的实例，只要调用 `add()` 、 `has()` 或 `delete()` 方法时传入了非对象的参数，就会抛出错误；

修改为：

> 1. 对于 `WeakSet` 的实例，若调用 `add()` 方法时传入了非对象的参数，就会抛出错误（ `has()` 或 `delete()` 则会在传入了非对象的参数时返回 `false` ）；

第八章 迭代器与生成器

生成器委托

> 生成器可以用星号（ `*` ）配合 `yield` 这一特殊形式来委托另一个生成器

修改为：

> 生成器可以用星号（ `*` ）配合 `yield` 这一特殊形式来委托其他的迭代器

解构与 `for-of` 循环

> Map 默认构造器的行为有助于在 `for-of` 循环中使用解构

修改为

> Map 默认迭代器的行为有助于在 `for-of` 循环中使用解构

第九章 JS的类

基本的类表达式

> 在匿名的类表达式内 `PersonClass.name` 是个空的字符串。而若使用了类声明，则 `PersonClass.name` 就会是 `"PersonClass"` 。
>
> > 使用类声明还是类表达式，主要是代码风格问题。相对于函数声明与函数表达式之间的区别，类声明与类表达式都不会被提升，因此对代码运行时的行为影响甚微。唯一显著的差异是匿名类表达式的 `name` 属性是一个空字符串，而类声明的 `name` 属性则与类名相同（例如，使用类声明时， `PersonClass.name` 的值为 `"PersonClass"` ）。

修改为（原作者删除了这两段的大量文字）：

> 使用类声明还是类表达式，主要是代码风格问题。相对于函数声明与函数表达式之间的区别，类声明与类表达式都不会被提升，因此对代码运行时的行为影响甚微。

继承内置对象

> 为了达成这个目的，类的继承模型与 ES5 或更早版本的传统继承模型有轻微差异，体现在以下两个重要方面：

修改为（删除最后一些文字）：

> 为了达成这个目的，类的继承模型与 ES5 或更早版本的传统继承模型有轻微差异：

此外删除后面两段文字的编号。

第十二章 代理与反射接口

起始第二段

> ES6 通过增加内置对象使得开发者能进一步接近 JS 引擎的能力。为了让开发者能够创建内置对象，

修改为：

> ES6 让开发者能进一步接近 JS 引擎的能力，这些能力原先只存在于内置对象上。

ownKeys 陷阱函数

此节的范例输出有误。

```
console.log(names.length);      // 1
console.log(names[0]);          // "proxy"

console.log(keys.length);      // 1
console.log(keys[0]);          // "proxy"
```

修正为：

```
console.log(names.length);      // 1
console.log(names[0]);          // "name"

console.log(keys.length);      // 1
console.log(keys[0]);          // "name"
```

此外删除这段话：

> 虽然 `ownKeys` 代理陷阱允许你修改少数操作所返回的键值，但它不能影响一些常用操作，例如 `for-of` 循环以及 `Object.keys()` 方法，这些都是使用代理所无法改变的。